RADC-TR-88-66
Final Technical Report
February 1989

SYSTEM ARCHITECTURE FOR FAULT-TOLERANT PROCESSES IN DISTRIBUTED SYSTEMS

University of Minnesota

Anand Tripathi, S. Azadegan, S. Ranka, S. Dong and V. Raghavan

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-66 has been reviewed and is approved for publication.

APPROVED: Mark F Zonca

MARK F. ZONCA
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific doucment require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-66
6a. NAME OF PERFORMING ORGANIZATION University of Minnesota 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) Computer Science Department 136 Lind Hall Minneapolis MN 55455	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0205
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62702F; PROJECT NO. 5581; TASK NO 21; WORK UNIT ACCESSION NO. PB
11. TITLE (Include Security Classification) SYSTEM ARCHITECTURE FOR FAULT-TOLERANT PROCESSES IN DISTRIBUTED SYSTEMS	
12. PERSONAL AUTHOR(S) Anand Tripathi, S. Azadegan, S. Ranka, S. Dong and V. Raghavan	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM Apr 86 TO Sep 87	14. DATE OF REPORT (Year, Month, Day) February 1989 15. PAGE COUNT 96
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Systems, Fault Diagnosis, Protocols, Replication, Fault Tolerance, Checkpointing
12	06		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The primary focus of this report is on system architectures and protocols for building fault-tolerant distributed systems. It addresses algorithms and protocols in four different areas: fault-diagnosis, error recovery in replicated systems, error recovery based on self-stabilization, and the use of masking redundancy in replicated systems using agreement protocols. This report is a collection of six technical papers that present the results obtained in this area. The first paper describes a system architecture for building resilient processes using replication and checkpointing. It describes the protocols for process replication management. The second paper presents an agreement protocol which provides the same view of the computation state to each correctly functioning copy of the process. The third paper presents a protocol for self-stabilization in binary trees. This protocol is a generalization of one of Dijkstra's protocols and for normal operations is sufficient to guarantee recovery from any erroneous state. The fourth paper presents a protocol for detecting the termination of a set of cooperating communicating processes. The last two papers address the problems related to

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Mark F. Zonca	22b. TELEPHONE (Include Area Code) (315) 330-2805 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

Block 19. Abstract (Cont'd)

fault-diagnosis in interconnected systems. The first presents a survey of the various fault-diagnosis algorithms based on the model proposed by Preparata, Metze & Chen (PMC Model). The second presents some results in direction of designing more efficient fault-diagnosis algorithms.

Table of Contents

Final Report

SYSTEM STRUCTURE FOR FAULT-TOLERANT PROCESSES IN DISTRIBUTED SYSTEMS

Summary

This is the final technical report on the RADC Post-Doctoral Program Contract F30602-81-C-0205, Task # B-6-3508 titled *System Architecture for Fault-Tolerant Processes in Distributed Systems*, funded through the University of Kansas Center for Research. The primary thrust of this project has been on the development of system architectures and protocols for building fault-tolerant distributed systems.

During the course of this effort we addressed various aspects related to building fault-tolerant distributed systems. The four basic functions required for building a fault-tolerant system are: error detection, fault diagnosis, fault isolation/reconfiguration, and error recovery/restart. Error detection involves detecting those states of the computation that do not satisfy the specifications. Fault-diagnosis algorithms identify the malfunctioning components in the system. Reconfiguration techniques isolate the faulty components from the rest of the system and possibly replace them with "healthy" units. Finally, the error recovery techniques bring the system back to some consistent state before restarting the operations again. In some systems a sufficient level of redundancy is provided to mask the effects of malfunctioning components. Such systems continue to function correctly even in the presence of some limited number of faulty components. In such systems one does not have to execute any fault diagnosis or error recovery algorithms given that there exist only some limited number of faulty components in the system. Such redundancy is called masking redundancy.

In this research effort we studied and investigated algorithms and protocols in four different areas: fault diagnosis, error recovery in systems with replication of processes or data, error recovery based on self-stabilization, and use of masking redundancy in replicated systems using agreement protocols.

This final report is a collection of six technical reports presenting the results of the research undertaken during the course of this contract. These reports are listed below:

(1) *Construction of Resilient Actions Using Replication and Checkpointing in Distributed Systems*, (A. Tripathi, S. Azadegan, and S. Ranka)

(2) *Constant Expected Time Randomized Byzantine Agreement Protocol without Shared Secrets and Cryptography*, (S. Ranka, A. Tripathi, and S. Azadegan)

(3) *A Protocol for Self-Stabilization in Binary Trees*, (S. Dong and A. Tripathi)

(4) *An Improved Algorithm for Termination Detection in Asynchronous Distributed Computations*, (A. Tripathi and S. Dong)

(5) *Fault Diagnosis in Distributed Systems -- Interim Report*, (V. Raghavan, S. Dong, and A. Tripathi)

(6) *Towards an Improved Diagnosability Algorithm*, (V. Raghavan and A. Tripathi)

The first report describes a system architecture for building resilient processes using replication and checkpointing. It describes protocols for managing a replicated object and the processes executing the actions invoked on the object. In our design one of the copies of a replicated object acts as the primary copy for executing a requested action. A process is created to execute a requested action, and periodically the execution state of the process is checkpointed with the other copies of the object, which act as the backup copies. In our design we have adopted an object-oriented view of distributed computing. Nevertheless, the protocols developed here are equally applicable to conventional process-oriented models of computing. In designing these protocols we first build a facility to detect if a site in the system is up or down. The protocols for checkpointing and recovery make use of this information. We assume fail-stop nature of site failures. The process replication protocols function correctly even in the presence of network partitioning, in which case every partition that has at least one available copy of the object will continue to function correctly. We have not addressed the problems related to merging or reconciling the copies when a partition is repaired.

The second report presents an agreement protocol that provides a consistent view of the computation to each correctly functioning copy of a replicated process. This protocol provides a mechanism to implement masking redundancy in the system. The algorithm developed here is probabilistic and does not require any shared secrets or cryptography.

The third report presents a protocol for self-stabilization in binary trees. The term self-stabilization means that if at any time the system is in some inconsistent state, then the normal execution of each process eventually brings the system back to some consistent state. Such systems do not require any special error handling protocols. The protocols for normal operations are sufficient to guarantee recovery from an erroneous state.

The fourth report presents a protocol for detecting the termination of a set of cooperating communicating processes. The interprocess communication is based on asynchronous message passing.

The last two reports address the problems related to fault-diagnosis in interconnected systems. The first of these two reports presents a survey of various fault-diagnosis algorithms based on the model proposed by Preparata, Metze, and Chien (PMC model). The last report presents some of the new results that we have obtained in the direction of designing more efficient fault-diagnosis algorithms.

Construction of Resilient Actions in Distributed Systems using Replication and Checkpointing

Anand Tripathi, Shiva Azadegan and Sanjay Ranka
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

1. Introduction

This report addresses the problem of building resilient actions and processes by replicating them to have a higher probability of surviving site crashes and completing successfully as compared to their non-replicated implementations. The ability of a system to continue its operations in spite of failures of some of its components is called the resiliency of the system. In this report we present a set of protocols for constructing such resilient operations in object-oriented environments by replicating the objects at multiple sites in a distributed system. We have used an object-oriented model of distributed computing; nevertheless, these protocols are equally applicable to the conventional, process-oriented, view of computation. The protocols described in this report facilitate checkpointing during executions of actions of a replicated object and restarting of an action at a different copy of the object when its primary execution copy crashes.

The primary contributions of this report are an interrupt-driven structure of the error recovery protocols in replication management and a system architecture which primarily uses the unreliable datagram facilities for normal operations and makes limited use of reliable message transmission protocols for transmitting exception conditions in the network during error recovery. We also present a novel protocol for status monitoring in the system using unreliable datagrams. One of the features of our design is that we do not require sites to have access to a secondary storage for recovery. This makes it ideal for applications where providing secondary storage is infeasible. Nevertheless, availability of secondary storage at certain sites will enhance the reliability of the system.

We assume that the distributed system environment underlying our protocols consists of fail-stop processors communicating across a communication network. The fail-stop assumption implies that a malfunctioning processor simply fails by crashing and does not behave maliciously or act as an adversary to

This work was supported by Rome Air Development Center under the Post-Doctoral Fellowship Program (Grant No. F-30602-81-C-0205).

the system. Thus the need of any Byzantine Agreement protocol [11] is eliminated in our design. The communication network provides a datagram facility which transports messages on the "best effort" basis, i.e., the delivery of messages is not guaranteed. We do not require the host sites to have access to secondary storage devices. The communication network is assumed to introduce delivery delays or loss of messages. In order to keep the primary focus of this report on the management of primary/secondary copies and recovery when the primary copy fails, we will assume that the communication network never gets partitioned. In the absence of this assumption we will have to incorporate some *network partition repair* protocols [6] in our designs.

In the past several systems have been designed to support replication of objects or processes. Some of the most noteworthy of these designs are Tandem's Guardian operating system [1] , ISIS [3], FTMP [9] and SIFT [13]. We have adopted several concepts from the designs of ISIS and Tandem's Guardian operating system. SIFT supports replication of processes for real-time applications and recovery from arbitrary failures; it uses Byzantine agreement protocols for inter-process communication between the replicated copies. ISIS uses a variety of reliable broadcast primitives [4] for replication management. In Guardian replication is limited to pairs only. We assume a reliable broadcast protocol as described in [5] only for communicating signals (exception conditions) during the recovery phases. Thus our design uses the more expensive reliable broadcast primitives only during the recovery phases for communicating signals.

In an object oriented environment each object comprising the system encapsulates some local data and a set of actions to manipulate the data. We might require that some of these objects, which might be of critical importance in the reliability and availability of the system, have higher resilience to error than the other objects. This can be achieved by replication of these objects at different sites. We use the concept of a *k–resilient object* [3], which is guaranteed to remain operational up to k site failures. We use the concept of *k–resiliency* in the context of resilient actions. In this report, we describe a scheme for establishing distributed checkpoints and recovery from site crashes that can be used to implement *k--resilient* actions.

The next section describes an abstract view of resilient actions (and objects) in our system. Section 3 outlines the requirement for achieving this functionality in replicated environments. Section 4 presents the functional layers in our design. It also outlines the important assumptions in our design. Section 6 describes the protocols for normal primary/secondary copy operation. Section 7 contains the protocols for

error recovery and the arguments for their correctness.

2. An Abstract View of Resilient Objects and Actions

Before discussing the issues involved in replicating an object and its actions, this section describes the logical view of a resilient object and its associated actions. Our design achieves this abstract view of an object by replicating it. Each object in an object-oriented environment encapsulates some data and a set of actions. To perform any execution on a particular object, the object is called to execute one of its actions. A result is returned after the action is completed. This action may be viewed as a sequence of synchronous operations $(a_1, a_2, \ldots, a_n)$. Each operation is a computation on one of the following types of data: (1) local data of the object, (2) temporary data of the object, (3) data of another object (in case of nested action). We will use the term "environment of an action" to refer to the temporary data maintained by the action and the state variables of the object to which that action belongs.

In the abstract view, each object has access to some k-resilient storage[1] which survives crashes of up

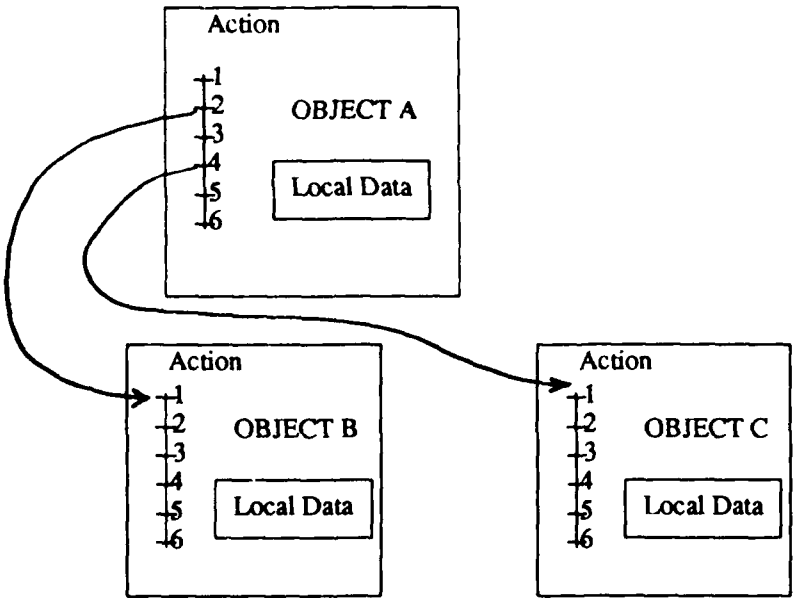


Figure 1: Abstract view of Resilient Objects

to *k* different processors or storage units in the system. This *k–resilient* storage is used for saving checkpoints during execution of an action. The following discussion presents this abstract view of a resilient object using an example. During normal operations, depicted in Figure 1, when an object A is called to perform Action 1, it will do a set of operations (1 to 6 in this example). In performing this invocation it will call object B and C to perform some actions. If the invocation is successfully executed, then the object will send some result to the object it was called from, i.e. client object. However in case of failure, object A may temporarily halt its execution. When the site comes up, based on its log information maintained on the k-resilient storage, it may restart its execution from some previous operation i.e. it may re-execute some operations. In order to cope with the consistency problems which may arise due to the re-execution of operations, we need that in case of failure, any action which may be re-executed on an object must be idempotent, so that when the same action is performed on an object it returns the same value,[2] without changing its local data. This can be achieved in two ways. The first way to achieve this is to undo all actions performed since the last execution of the action, and then execute the action again. Thus performing the action again returns the same value and has the same effects on the local data as before. The second way is to retain the results of the action performed on the object long enough, so that if a previously executed actions on the object is asked to be performed again, the retained value can be sent to the client object. This will retain the idempotency.

Let us now consider a recursive call, as shown in Figure 2, on object A. Object A executes an action which results in a chain of invocations on other objects(the length of which is nil in case A calls itself directly), the last of which calls A again. In such a case, the state of local data of object A, which may be modified, should be the state in which last call was made from object A. This state can be easily identified by looking at the call-id of the recursive call. In our design, a call-id for any action invoked by an object is obtained by concatenating the call-id of the action this object was invoked with, its object UID, action-id and operation number. The call-id for the top-most level action is a globally unique identifier. In this example object A calls another object B in operation 4 of Action 1 and object B calls object C and object C in turn calls object A. Now the state of object A which this invocation of operation on A by C should see

[1] Such a storage can be implemented using either some disk storage devices or the volatile storage of some other sites in the system.

[2] One should note here that such a model will not applicable in systems where the results of some invocations are time-dependent.

is the state in which object A was left when the operation 4 was performed.

In the above examples we described some characteristics of an action performed on an object. We want these characteristics be preserved when one or more of the objects is replicated, i.e. whether an object, on which an action is invoked, is replicated or non-replicated is transparent to the client object. In the later sections, we describe a scheme on how to replicate objects and their actions preserving the above required characteristics.

3. Replicated Environment

In a replicated distributed environment an object can continue execution of its actions despite site failures. For achieving this objective, information about the execution must be distributed among different processes as the execution proceeds so that another copy of the process, at another site, can take over and continue the operation when the executing copy fails. Moreover it is necessary that copies of an object be consistent in performing further actions. This requirement is important for a consistent behavior of all copies as a single object.

In a replicated object-oriented environment an action is performed, as in the non-replicated case, by invoking an action on a replicated object. In fact, it is totally transparent to the invoker that the called object is replicated. These actions are guaranteed to be executed atomically. The operating system underlying our design supports locating objects (or some of the copies of a replicated object) and delivering the invocation messages to them. In addition, these objects can be accessed concurrently, which requires some concurrency control mechanism in order to preserve the object consistency. Many such mechanism have

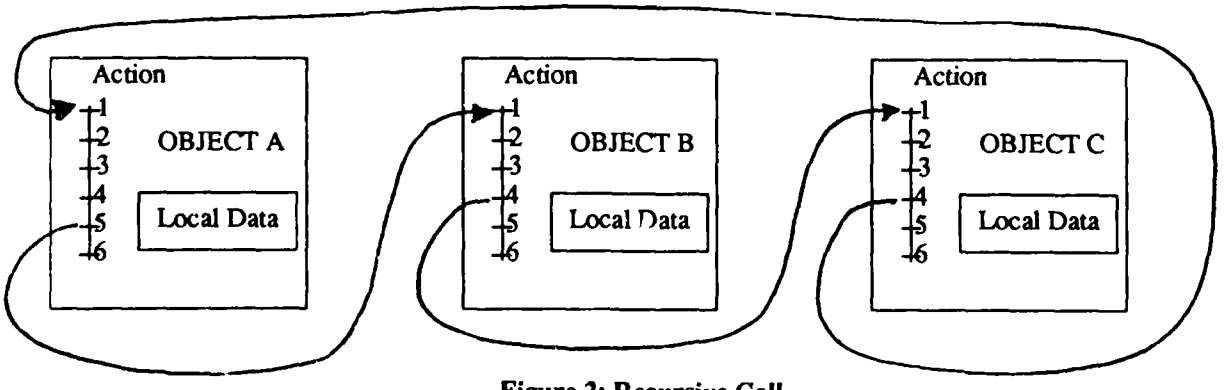


Figure 2: Recursive Call

already been developed [2] and we will not discuss any of these here. Our scheme is orthogonal to any specific concurrency control mechanism.

When an action on an object is invoked, we must determine which copies of the object will start the execution of the action. We refer to such copies as the primary copies for that invocation. We can either have multiple copies of an object executing the same action concurrently or only a single copy executing the action. We chose to have a single copy executing as we believe that multiple concurrent executions of an action is a wastage of system resources and it does not increase the availability of the system in a very significant way. In some situations one might require to execute more than one copy concurrently. In the scheme which we describe here no consistency requirement will be violated even if we allow simultaneous execution of multiple copies. Since we chose to have a single copy executing, we must determine which copy should start the operation. There are two possible cases: either both the invoker and invoked object have copies on the same site or they have copies on different sites. In the former case, it is logical to choose the site for the primary copy as the one that hosts both objects. In the latter case, the site can be chosen either based on some criterion such as of CPU utilization of the site or physical distance of the site to the invoker, etc., or the site can be chosen statically by having a default primary copy for each action. We chose to have a default primary copy, which initiates the execution, when both the invoker and invoked objects do not have copies on the same site. Once the primary copy of an action starts the execution, it periodically sends synchronous and asynchronous checkpoint information to all secondary copies. If the primary copy of an action fails, then the recovery protocol is initiated to elect a new primary copy.

4. System Architecture for Replication Management and Recovery

An abstract view of the system architecture is shown in Figure 3. The top-most layer represents the processes executing the currently on-going actions of a replicated object by its copies at different sites. Such a process may be either in the normal computation (i.e., executing the action as the primary or secondary copy) or in the recovery phase (i.e. executing election protocol). The primary focus of this report is on the protocols executed by these processes. In our model each site has only one copy of the object.

Underlying these processes, corresponding to each copy of the replicated object, there is an object manager process which is responsible for scheduling these processes, enforcing necessary concurrency

control protocols, and interrupting these processes when some exception conditions (signals) arise. Corresponding to every executing action at a copy of a replicated object, there is a process which executes that action. We refer to this process as the primary/secondary copy of that action. The object manager also coordinates with the object managers at other sites to maintain the current status of the replication configuration, i.e. the status of the other copies as *up* or *down*. For this purpose each object manager periodically executes the *status monitoring protocol* in which it broadcasts status information to other object managers which are participating in the same action; we will denote this period by T. In this protocol we also assume that the clocks in the network are synchronized using some network clock synchronization algorithm[10]. The following properties of this *status monitoring protocol*, the details of which are given in the Appendix, are of interest to us in this design. (1) If a copy crashes during the interval $[kT, (k+1)T]$, then other *up* copies will detect this failure by the time $(k+3)T$. (2) If a copy restarts during the interval $[kT, (k+1)T]$, then with a very high probability[3] all other *up* copies will know about this restart by the time $(k+3)T$. (3) If at time kT a copy incorrectly assumes some other copy to be down, then there is very high probability that this incorrect view will get corrected by time $(k+1)T$.

With every object manager some configuration related data is maintained. This includes list of the unique identifiers (UIDs) of the other copies, list of copies which are currently available, default primary copy, the current operation being performed and their primary copies (Figure 4). Moreover, for every replicated object there is a static ordering of copies, called the *election manager list (EM_list)*, which is used to designate one of the copies as an *election manager* to elect a new primary copy for an on-going invocation, in case of primary copy failure. The election manager elects one of the available copies, and hopefully the one with the latest checkpoint, as the new primary copy.

Our protocols are interrupt-driven in the sense that the execution of a statement during the normal computation phase or during the recovery phase may get interrupted if any of the specified exception conditions arises. An exception condition is *enabled* if an exception handler is associated with any statement or any of its outer blocks. Reception of a signal, for an enabled exception condition, at an object may cause interruption of some of its currently executing actions. we have referred to signals, such as

[3] This probability is a function of the reliability of the underlying datagrams communication network.

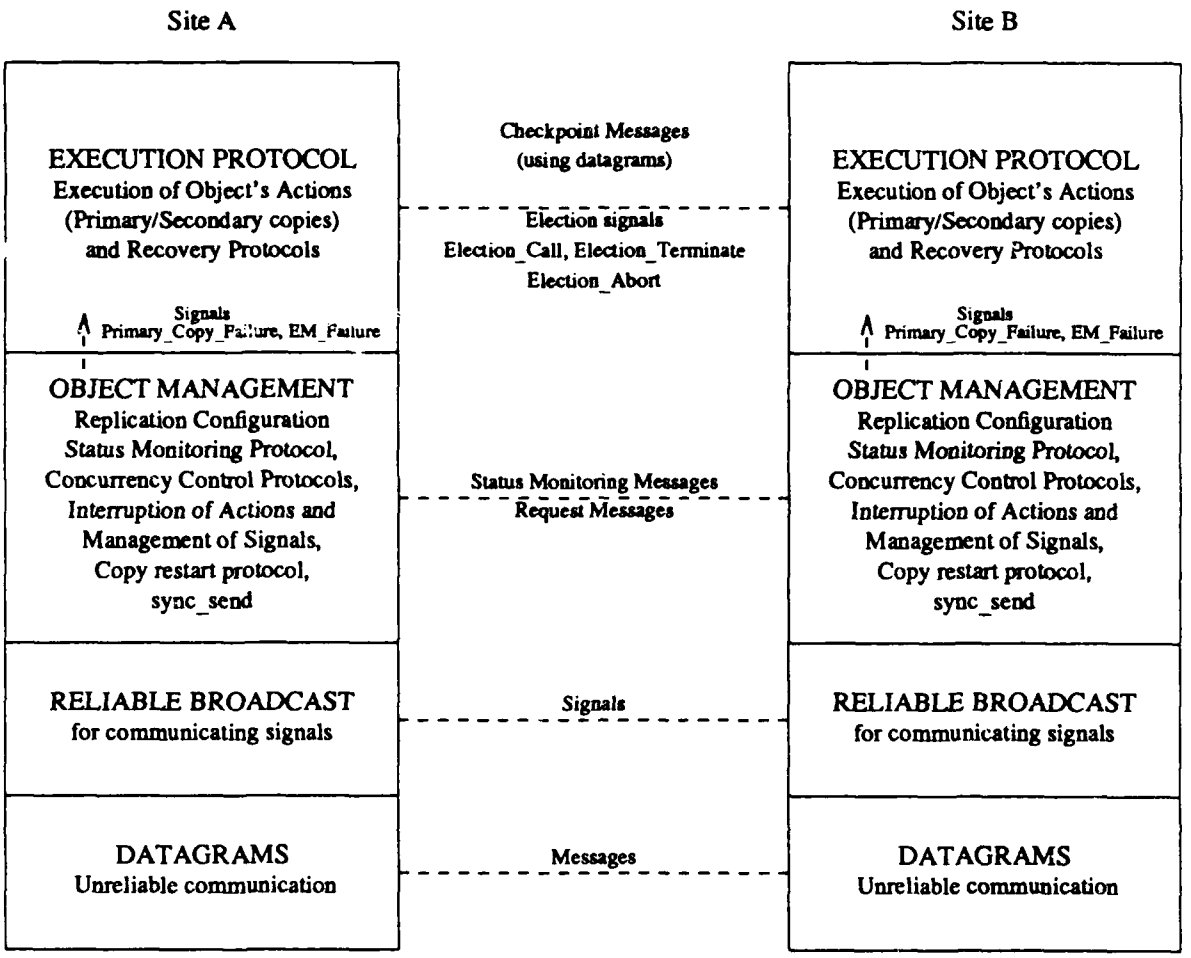


Figure 3: System Layers of Abstraction

Election_Call, Election_Termination, Election_Abort, as global signals since they are sent from one site to another site; in other words they are sent through the reliable broadcast system. Local signals, such as *EM_Failure* and *Primary_Copy_Failure*, are those which are generated by the object manager as the result of detecting some failures. The local signals are sent only to the local processes of the object. In contrast to signals, the arrival of a message does not cause any interrupt at the destination object.

Whenever an object manager detects (by executing status monitoring protocol), that some other copy x has gone *down*, it updates its configuration data. It also checks whether x had been the primary copy of some on-going actions, in which case it sends a signal called *Primary_Copy_Failure* to the local processes executing those actions as secondary copies. This signal will cause the secondary copies to execute the recovery protocol (election protocol). Similarly, if x was the election manager copy for some action, an

exception condition called *EM_Failure* is signaled to the local process, corresponding to that action, executing the recovery protocol. When an election manager copy starts election, it signals an exception condition called *Election_Call* to interrupt all other copies to participate in the election. This signal causes all processes (at those copies) executing as secondary copies to participate in electing a new primary copy. Similarly, the successful completion of an election is signaled as a condition called *Election_Terminate*. An unsuccessful completion of the election is signaled using *Election_Abort*.

The underlying communication system provides an unreliable, asynchronous message passing primitive *send(msg) to destination* which sends the message *msg* to the object (or objects) specified in the *destination* field. Also it provides a primitive *receive(msg) from source* for receiving a message *msg* from the object (or objects) specified in the *source* field. In this model *receive* is a blocking operation, i.e. a call to this function returns if and only if expected message(s) is(are) received, and in such a case this function returns true. The source and destination are specified as the unique identifiers (UIDs) of the objects. In case of replicated objects, a specific copy is specified by extending the unique identifier with the ID of the host site of that copy, called *Extended_Object_UID*, (assuming that only one copy resides at any host site). The *send* primitive is implemented using the datagram facility which does not guarantee the delivery of a

```
/* List of the copies of the replicated object, and their status */
Configuration : list of (copy : Extended_Object_UID;
                         status : (up, down));

avail_copies : list of copies in the Configuration list with up status;

/* Static list of copies which can act as election managers */
EM_List: list of Object_UID; /* object's copies */

/* Current calls being executed on the object */
Current_Call_Table  : list of (Call_id :     String;
                               Action_id     : Integer;
                               Initial_version: Memory_address;
                               Latest_chkpnt  : Integer;
                               Current_version: Memory_address;
                               Primary_Copy   : Extended_Object_UID)

/* Retained values of the different calls performed on the object */
Retained_values     : list of (Call_id : String;
                               Response: Message_type
                               )
```

Figure 4 : Data Structures Maintained with an Object

message.

Global exception conditions (signals) are communicated by a copy of the replicated object to another copy (or copies) using the primitive *signal(condition) to destination* which raises the specified exception condition at the objects specified by the *destination* field. It is possible to pass some parameters to the destination object when signaling an exception condition. We assume here that the signals are communicated using the reliable broadcast protocol described in [5]. This broadcast protocol has the following properties: (1) All objects in the destination list receive the signal. (2) The order in which signals are sent by a sender is preserved at the receivers. (3) Signals sent by different senders are received by all common receivers in the same order. We assume that the reliable broadcast protocol tries to persistently deliver signals to only those copies which remain *up* during the execution of the broadcast protocol and ignores any copy which goes *down*. For the correctness of our recovery protocols we require that the status broadcast period T be selected such that all signals are guaranteed to be delivered to all the *up* copies within $2T$.

In this report we do not discuss the *copy restart protocol* which is executed by the object managers when a failed copy tries to rejoin the configuration. To ensure the correctness of the election protocol in our design, we assume that the restart protocol has the property that it delays inclusion of a restarting copy in the configuration if a recovery (election) is currently on-going for some action. Also, if the permission to join the configuration is signaled to a restarting copy during the interval $[kT,(k+1)T]$, then the latest by the time $(k+2)T$ this copy will start sending its status messages to all other copies. Which means that with very high probability all other copies will include this copy in their *avail_list* by time $(k+4)T$. A new (or restarted) copy does not act as an intermediary to include other new copies in the configuration during this $4T$ period.

5. Notation for Protocol Descriptions

All protocols are presented in a Pascal-like notation. Most of the constructs have conventional meaning. The **loop...end** construct represents iteration of the enclosing sequence of statements; the iteration is terminated with the execution of an **exit** statement. We also use an exception handling model with the statements to provide an interrupt driven execution of the protocols.

We use the termination model of exception handling[12] to describe the protocols. Similar models have been used in several programming languages such as Ada[8], Gypsy[7], and CLU[12]. In our model an exception handling block can be associated with any program statement for dealing with anticipated exception conditions. The exception handling block lists the exception conditions and the actions (called exception handlers) associated with them. If any of the listed exception conditions arise during the execution of its associated statement, the corresponding exception handler is executed and the execution of this handler terminates the execution of that statement. For example, in the program fragment shown below, a *timeout* condition will terminate the computation enclosed within the **begin..end** block and execute the actions associated with the timeout condition. The *set_timer(timeout_period)* construct is used to set the timer with the desired timeout value; this will cause raising of the timeout exception after the expiration of the specified units of time.

```
set_time(100);
begin
   S1; S2;.... ; Sn;
end when
          C1: X1;
          C2: X2;
          C3: X3;
          timeout: /* execute the timeout action */
      end;
S;
```

In the above example the execution of the sequence of statements *S1;S2;...;Sn* will be interrupted and terminated if any of the exception conditions *C1*, *C2*, *C3* or *timeout* is signaled. If more than one conditions are signaled, then the exception handler corresponding to only one of the signaled conditions will be selected and executed. This selection is non-deterministic. After executing the exception handler, the execution will proceed with the next following statement, which is *S* in the above example. If none of the specified exception conditions arise during the execution of the sequence *S* 1;.....;*Sn*, then all exception handlers are ignored and the execution proceeds with the next statement.

We also use the **when...end** construct to wait concurrently on some number of exception conditions or message reception events. The execution of such a statement completes when any of the specified condition arises; in such an event the corresponding exception handler is executed. If more than one conditions arise concurrently, then one of them is selected non-deterministically and the corresponding exception

handler (which can be a null statement) is executed. For example:

```
when
    C1: X1; /* C1 and C2 are some exception conditions */
    C2: X2;
    receive (msg) from a: X3;
    timeout: ;
end;
```

The execution of the above statement will complete if any of the conditions *C*1, *C*2 or *timeout* is raised, or a message is received from *a*, and the corresponding exception handler is executed. In this example, in case of *timeout*, no action is taken for exception handling. In case of nested blocks, the exception conditions defined with the outer blocks always have priority over the ones associated with the enclosing blocks.

Sometimes it is required that all external exception handling (i.e., defined with the outer blocks) be masked and kept pending during the execution some sequence of statements. As a notation we enclose such a sequence of statements within square brackets "[" and "]". This only means that any exception handling associated with the outer blocks will not be effective during the execution of this sequence. If any of the statements in this sequence have exception handling associated with them, then those exception handlers will still be enabled and executed when those conditions arise. (This makes the execution of the sequence atomic with respect to outer exception conditions.)

Using the unreliable communication primitives described earlier we implement a new primitive *sync_send* as described in Figure 5. The *sync_send* primitive persistently tries to send a message to each of the destination objects until a response is received from all the objects which appear to be up; it ignores any destination objects which goes down during the broadcast (this *sync_send* primitive should not be confused with the reliable broadcast protocol assumed for sending signals). When *sync_send* is invoked, a procedure can be passed to it through the formal parameter *ack_handler* to handle the response messages from the destination objects.

6. Replication Management Protocol

This section describes the Execution Protocol, shown in Figure 6, executed by each of the replicated copies. A copy can function as a primary copy (executing the protocol shown in Figure 7) or as a secondary copy (executing the protocol shown in Figure 8) and this role can change dynamically due to the

```
Procedure sync_send(msg: message; dest: list of Object_UID, ack_handler)
begin
  All := [ dest ];
  loop
     if All = ϕ then exit else;
     send(msg) to All;
     set_timer(timeout_period);
     loop
        if All = ϕ then exit;
        when
           receive( ack ) from some a in All:
              All := All -[a];
              ack_handler(ack, a);
           down(a) for some a in All: All := All - a;
        end /* of when */
     end /* of loop */
     when
        timeout:;
     end /* of when */
  end; /* of loop */
end; /* of sync_send */
```

Figure 5: Synchronous Send

failure and recovery as determined by the execution protocol described below. The description of the protocol is presented here in terms of its four major components -- protocols for primary copy, secondary copy, election manager, and election participant. When an action is invoked, after the execution of appropriate concurrency control protocols and designation of the primary copy ,the action is scheduled for execution. All the *up* copies of the object execute the Execution Protocol described in Figure 6. The invoked action is passed as a parameter to this procedure. A copy can determine whether it is the primary copy or a secondary copy for the action by executing the function *I_AM_PRIMARY* (*A*). It then executes the appropriate protocol as described below. One must note here that a copy can be concurrently acting both as the primary copy for some invocations and as the secondary copy for the others. This feature can be effectively used in incorporating various load balancing protocols in our design. The execution of the primary or secondary copy protocol can get interrupted if the local signals *Primary_Copy_Failure* or *Election_Call* are raised. Under these conditions, the exception handler for the condition is executed which essentially forces each copy to participate in the Election Protocol to elect the new primary copy. It is possible for the execution of the Election Protocol at a copy to get interrupted if any of the conditions *EM_failure* or *Election_Terminate* is raised. After such an interruption and the execution of

corresponding exception handler, the election protocol continues until the new primary copy is known.

6.1. Primary/Secondary Copy Execution

When an object starts executing an action, the primary copy for that action first performs a synchronous checkpoint with the secondary copies. This informs all other (secondary) copies that a call was made with a particular unique number for performing an action. Moreover all the copies should be in the same state to perform that action, so appropriate checkpointing is also required.

After performing the above steps, the primary copy executes action in the same fashion as an unreplicated object, but periodically sends synchronous or asynchronous checkpoints to other copies (see Primary Copy Execution protocol in Figure 7). Such checkpoint operations are inserted in the code by the programmer. All checkpoints messages are sent using the unreliable datagram facility. All checkpoints are numbered in the increasing order starting with initial checkpoint as 0. A checkpoint message contains complete state of the object's local data which has been modified by the primary copy. A secondary copy

```
Procedure Execution_Protocol (A: Action);
var
 crash_restart: Boolean; /* indicates whether a crash has occurred during the execution of this action */

Procedure Primary_Copy_Protocol(A: Action);  /* Figure 7 */
Procedure Secondary_Copy_Protocol(A: Action); /* Figure 8 */

begin
  crash_restart := false;
  loop
     begin
        if I_AM_Primary(A) /* This function is true iff this copy is currently the primary copy */
        then Primary_Copy_Protocol(A);
             exit; /* completion of the primary copy execution */
        else Secondary_Copy_Protocol(A);
            exit; /* completion of the secondary copy execution */
     end when
              Primary_Copy_Failure, Election_Call:
             /* detection of primary copy failure or an election has been started */
                   primary_copy := unknown;
                   Election_Protocol;
                   crash_restart := true;
          end; /* of when */
  end;
end; /* end of Execution_Protocol */
```

Figure 6: Execution Protocol

receiving a checkpoint is not required to wait for any previous checkpoint messages except the initial message. In order to preserve the correctness of checkpoint information kept by secondary copies ,the exception conditions are masked while a secondary copy is storing the received information. In case of synchronous checkpoints, the primary copy waits for all the other copies to respond before it continues the execution of the action. However in an asynchronous checkpoint, the primary copy simply broadcasts the checkpoints and continues the execution of the action. Thus if the primary copy does not go down, the action is performed almost in the same way as in the case of an unreplicated object.

After the end of all the operations, a final synchronous checkpoint is performed. In this checkpoint all the final values of the modified local data and the results to be sent to the invoker are recorded by other copies. In case of the primary copy failure during any of the three phases, a new primary copy is selected

```
Procedure Primary_Copy_Protocol (A:Action);

begin

 I: if not crash_restart
    then
        /* Establish the first checkpoint */
        sync_send(<Call_id,Cur_version,Initial>, avail_copies, ack_hdlr);
    else
        /* load the latest checkpoint information */

II:     /* Perform the action */
            loop
                    if end_of_action(A) then exit;
                    case operation of
                            A_checkpoint : /* Establish an asynchronous checkpoint */
                                            send(<A_chkpnt, Call_id, Cur_version, chkpnt#>) to avail_copies;
                            S_checkpoint : /* Establish a synchronous checkpoint */
                                            sync_send(<S_chkpnt, Call_id, Cur_version, chkpnt#>, avail_copies, ack_hdlr);
                            Otherwise : [ /* Execute the requested operation with all exceptions masked */ ]

                    end;
            end;

  III:  /* Establish final checkpoint */
            sync_send(<Call_id, Cur_version, Final>, avail_copies, ack_hdlr);
            /* Send the result to the invoker */
            sync_send(<result>, invoker, ack_hdlr)

end; /* of Primary_Copy_Protocol */
```

Figure 7: Primary Copy Execution Protocol

```
Procedure Secondary_Copy_Protocol (A : Action);

begin
  loop
      receive(Chkpt_msg) from primary_copy
      [ case Chkpt_msg.chkpnt_num of
          Initial : /* add the received information to current call table for the specified action */
                        Call_id := Call_id;
                        Initial_version := Address(cur_version);
                        Primary_Copy := Sender_id;

          Otherwise : /* update the current call tbl for the given call-id */
                        if chkpnt_num > Latest_chkpnt
                        then
                              Latest_chkpnt := chkpnt_num;
                              current_version := Address(cur_version);
      end; ]
      /* send an acknowledgement back */
      if S_chkpnt then send(ack) to Primary_Copy ;
      if chkpnt_num = Final then exit;

  end;
end; /* of Secondary_Copy_Protocol */
```

Figure 8: Secondary Copy Execution Protocol

by executing the election protocol (described in Figure 9) and a new primary copy continues the execution from the latest checkpoint. Thus a replicated object does not stop its execution in case of site failures and the execution of its actions is guaranteed to complete as long as an active copy exists.

In case of a recursive action call to the object, we ensure that the copy which starts the execution is the same as the active copy of the parent operation of this call. If the primary copy of the parent operation had failed, the call will be blocked till the new primary copy reaches the state in which the parent operation was executed; such a state can be easily identified by looking at the call-id of the recursive call. This keeps the behavior of the replicated object the same as that of an unreplicated one, in case of recursive calls.

When an object is acting as a server object, it retains the result message sent to the client for an invocation. This is needed as the client might be replicated and in case of failure of one of its copy, another copy may re-execute the invocation operation and call the server to perform the same action. We have adopted this idea from the ISIS design. Thus whenever a call is made to an object to perform certain action, it checks whether the operation was performed earlier. If it was performed earlier, then it simply sends the retained value for that action. Hence these results must be stored by the server object till it is sure

that the client object can never make a call again. This can either be done when the client object has completed all the operations for its action, and has stored its own result with the other copies. However this may not be very efficient. A more efficient way would be to store results for a certain duration and discard them after some long time. This time should be large enough so that the operation cannot be performed again.

6.2. Recovery Protocols -- Election of A New Primary Copy

If the primary copy of an invocation fails, the other copies are informed of this failure by their object managers. The execution of the secondary copy protocol may get interrupted if either *Primary_Copy_Failure* or *Election_Call* signal is received (see the exception handling part in Figure 6). A secondary copy execution when interrupted because of these signals executes the Election Protocol (Figure 9). In case of a secondary copy failure, other copies are informed and they just need to update the list of available copies. Among all the copies executing the Election Protocol one of the available copy with the highest priority (according to the *EM_list*) acts as the election manager and executes the Election Manager Protocol (Figure 10); all other copies execute the Election Participant Protocol (Figure 11). The function *I_AM_EM* returns true if and only if the copy executing this function is eligible to act as the election manager. After the completion of election protocol a new primary copy is designated to carry on the execution.

The Election Protocol execution may get interrupted if an *EM_failure* or *Election_Termination* signal is received. In case of the first signal, a copy restarts its Election Protocol and once again checks if now it should be the new election manager, and then accordingly executes either the Election Manager Protocol or the Election Participant Protocol. In case of *Election_Termination* signal, the id of the new primary copy is sent to all copies through the signal, the execution of the Election Protocol is terminated, and the Execution Protocol continues executing either the Primary Copy Protocol or the Secondary Copy Protocol.

The election protocol is structured such that if due to some error conditions two copies start executing the Election Manager Protocol, then only one of them will be able to complete the election successfully. The three-phase structure of the election protocol is very similar to that of reliable broadcast protocols described in [5]. A copy acting as the election manager first sends a signal called *Election_Call* to all

```
Procedure Election_Protocol;
var
   new_primary : Object_UID;
   successful  : Boolean;
   EM : Object_UID;  /* Current Election Manager copy */

Procedure Election_Manager_Protocol; /* Figure 10 */
Procedure Election_Participant_Protocol; /* Figure 11 */

begin
    loop
        begin
            if I_AM_EM
            then
                Election_Manager_Protocol;
                if successful
                then exit /* election successfully completed */
             else
                 Election_Participant_Protocol;
                 exit;
         end when
                 EM_Failure: ;
                 Election_Termination ( Primary_Copy ): exit;
               end; /* of when */
    end;
end;
```

Figure 9: Protocol for Electing New Primary Copy

other available copies to make sure that all those copies are also ready to participate in the election. This signal is sent persistently until an acknowledgement is received from all the other copies. An election participant sends an ack_1 only if it has not received an *Election_Call* signal from some other higher priority copy acting as an election manager. Otherwise it sends back a *nack*. When a participant sends an ack_1, it sets its *EM* variable to the id of the copy which sent this *Election_Call* signal. If in the first phase any *nack* is received by the Election Manager procedure, its execution terminates setting the global variable *successful* to *false*.

In the second phase, the copy executing as an election manager and receiving all ack_1 messages in the first phase requests the latest checkpoint numbers from all the participant copies. Meanwhile if some participant has received an *Election_Call* from some other higher priority copy, its *EM* would be set to the id of that copy. Therefore, in response to the latest checkpoint request, a participant copy sends the latest checkpoint only if the requester's ID is equal to *EM*; otherwise it sends a *nack* The Election Manager Protocol terminates with *successful* set to false if any *nack* is received during this phase.

```
Procedure Election_Manager_Protocol;
var
   no_nack_received : Boolean;
   participants     : list of Extended_Object_UID;
begin
  begin
        participants := avail_Copies;
        successful := true;
        signal(Election_Call) to participants;
        set_timer(timeout_period);
        remaining_copies:= participants;
        StartTime := clock;
        loop                                          /* Phase I */
            when
              received(ack) from a, down(a):
                 remaining_copies := remaining_copies - [a];
                 if remaining_copies = ϕ then exit;
              time_out:
                 if remaining-copies ≠ ϕ
                 then
                    signal(Election_Call) to remaining_copies;
                    set_timer(timeout_period);
                 else exit
          end; /* of when */
        end; /* of loop */

        participants := participants ∩ avail_copies;          /* Phase II */
        if no_nack_received
        then
           sync_send('latest_chkpnt?',participants ,elect_primary)
           if no_nack_received
           then [ /* mask exception handling from outside blocks */
              sync_send('Election_Success',participants,all_acks);     /* Phase III */
              if no_nack_received
              then
                   when
                      (clock - StartTime) ≥ 4T :
                   end;
                   signal(Election_Terminate) to avail_copies; /* signal to all up copies */
              else
                     successful := false;
                     signal(Election_Abort) to avail_copies; ] /* resume exception handling */
           else successful := false;
        else successful := false;
  end when
         Election_Call from a where priority(a) > priority(mycopy):
                EM := a; send(ack1) to a; successful := false;
     end;
end; /* of Election_Manager_Protocol */
```

Figure 10: Election Manager Protocol

The Election Manager Protocol enters the third phase if no *nack* is received during the second phase. In this phase, the election participant copy with the largest checkpoint is elected as the primary copy. It then sends a message *'Election_Success'* to all the participants. A participant which still has its *EM* set equal to the id of the sender of this message sends an ack_3 message, otherwise it sends a *nack*. Up to the point of sending this ack_3 message, a participant is free to send an ack_1 or the latest checkpoint number to any other higher priority copy which might start executing the election manager protocol. However, once the ack_3 in the third phase is sent, a participant has to wait for either the *Election_Abort* or the *Election_Termination* signal and during this period it may not send ack_1 messages to any other election manager. Therefore no *Election_Call* signal is accepted in the third phase.

Acceptance of an *Election_Termination* signal clears all pending *Election_Call* signals. Similarly the acceptance of an *Election_Call* signal clears all pending *Election_Termination* or *Election_Abort* signals. If the election completes successfully, the Election Manager Protocol sets *successful* to *true* and that

```
Procedure Election_Participant_Protocol;
begin
  EM := highest priority in (avail_copies ∩ EM_List);
  loop
     when
        Election_Call from copy a:
              if a has the highest priority in (avail_copies ∩ EM_List)
              then EM := a;  send (ack1 ) to EM;
              else send ( nack ) to a;
        receive( 'Latest_chkpnt?' ) from a:
              if EM = a
              then send( Latest_chkpnt ) to EM;
              else send( nack ) to a;
        receive ( 'Election_Success' ) from a:
              if EM = a
              then
                send( ack3 ) to EM;
                when
                  Election_Terminate( Primary_Copy ): exit; /* the loop and terminate election */
                  Election_Abort from EM:
                end;
              else send( nack ) to a;
     end /* of when */
  end; /* of loop */
end; /* of Election_Participant_Protocol */
```

Figure 11: Election Participant Protocol

causes the termination of the Election Protocol at the election manager copy. Whenever the Election Manager Protocol terminates with *successful* equal to *false*, the Election Protocol is retried.

One should note here that a successful election is forced to last at least $4T$ units of time. Also, the election termination signal is sent to all available copies and not just to those which participated in the election. This makes sure that any copy which was given permission to join the configuration before the start of the election will also receive this signal. Because of the $4T$ duration, all such new copies will be in the available list when the termination signal is sent.

It is possible that the an execution of the Election Manager Protocol may get interrupted if it receives an *Election_Call* signal from some other higher priority copy. In this case the Election Manager Protocol terminates by sending an ack_1 message to the sender of the signal, and setting *successful* to *false* and *EM* to the id of the sender of the signal. (See the exception handling part of Figure 10.) However, during the third phase all exception conditions associated with the outer blocks are masked. This means that once an election manager enters the third phase, an *Election_Call* signal from some higher priority process will be kept pending during this phase. If the election completes successfully then all copies including the waiting election manager will also receive the *Election_Termination* signal and all pending *Election_Call* signals will be cleared.

7. Correctness of the Election Protocol

In order to show that this recovery protocol functions correctly we need to show that this protocol is (1) free from deadlocks and livelocks, (2) once an election is started eventually only one *Election_Termination* signal will be generated, and (3) the same election signal will terminate the election protocol at all participant copies.

The absence of deadlocks in the election protocol follows from the following observations. Only in the following situations can a copy hold sending a response message to another copy and cause it to wait. (1) A participant which has sent an ack_3 message to an election manager waits for either an *Election_Termination* or *Election_Abort* signal and does not send any response to other election managers which have issued either an *Election_Call* or the latest checkpoint request. (2) An election manager which is in phase III holds responding to any *Election_Call* signal from some other election manager. (3)

An election manager in phase III holds sending either the *Election_Termination* or *Election_Abort* signal until it has received the response messages from all participants in the protocol. Note that a participant never holds response messages to an election manager which is in the third phase. To show absence of deadlocks we show that only a higher ordinality process can hold sending responses to a lower ordinality process. Assign a time-dependent ordinality to a process as a pair of integers $<p,q>$ where p is equal to the current phase number for an election manager process and for a participant it is the highest phase number messages it has received from and responded to any election manager. q is equal to 0 for a participant and 1 for an election manager. The value of this number is interpreted as the lexicographic ordering of p and q. Thus an election manager in phase III has ordinality 31 and a participant which has sent it an ack_3 message has ordinality 30.

The proof of absence of livelocks is based on the observation that the sequence of values assigned to the variable *EM* of any process has monotonically increasing values (in terms of priorities assigned to the election managers in the *EM_list*), unless an election manager goes down. Thus if two or more election managers repeatedly execute their protocols, then eventually all participants will have their *EM* set to the id of the highest priority election manager which does not fail. Eventually only that election manager will complete the election successfully and generate the *Election_Termination* signal.

Now we want to show that all participants will terminate their election because of this signal. The only other possibility for a participant to terminate the election protocol with its current election manager (which is in its third phase) is when it receives the *EM_failure* signal. If the election manager fails immediately after signaling *Election_Termination*, then all participants will receive this signal with in the next T period; moreover, the *EM_failure* condition will be detected and signaled only $2T$ units of time after the failure event. Thus all participant will terminate the election successfully due to the same election termination signal.

Appendix: Status Monitoring Protocol

This protocol is used for monitoring the *up/down* status of some set of processes in the network. The protocol assumes that the clocks in the network are synchronized using some network clock synchronization protocol. The protocol executes in synchronous steps; at every T interval, each process broadcasts its

view of the status information to all the other processes in the network. (Assume that T >> the maximum message delay in the network.) This status information consists of status of other processes from the view point of this sender. For each process, the status is maintained in terms of three colors: white, grey, and black. White and grey colors are interpreted as *up* status for that process, and black implies *down*. During phase k, a process u computes the status of the some other process v based on the status messages it received from other processes in phase $(k-1)$. After computing this new status, it broadcasts it all other processes.

Phase 1:

```
/* mark status of every copy in the configuration as grey */
send(status_info) to others
```

Phase k>1:

```
for all v in configuration
   do
         if a status message was received from v in phase k-1
         then mark v as white
         else if any status message (received in phase k-1) had v marked as white
            then mark v as grey
            else mark v as black
   end
send(status_info) to others
```

References

[1] Joel Bartlett, "The Nonstop Kernel," *Proc. of the Eighth Symposium on Operating Systems Principles*, pp. 22-29 (December 1981).

[2] P.A. Bernstein and Goodman, Nathan, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys, Vol. 13, No. 2*, pp. 185-222 (June 1981).

[3] K.P. Birman, T.A. Joseph, T. Raeuchle, and A.E. Abbadi, "Implementing Fault-Tolerant Distributed Objects," *IEEE Transactions on Software Engineering* **SE-11, Number 6** pp. 502-508 (June 1985).

[4] Kenneth P. Birman, "Replication and Fault-Tolerance in the ISIS System," *Tenth ACM Symposium on Operating Systems Principles*, pp. 79-86 (December 1985).

[5] Jo-Mei Chang and N.F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems* **Vol. 2, No. 3** pp. 251-273 (August 1984).

[6] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen, "Consistency in Partitioned Networks," *ACM Computing Surveys* **Vol. 17 No. 3** pp. 341-370 (September 1985).

[7] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, and D.F. Hare, *Report on the Language Gypsy, Version 2.0*, Institute of Computing Science, The University of Texas at Austin, Austin TX 78712 (September 1978).

[8] Honeywell, *Reference Manual for ADA Programming Language* July 1980.

[9] A.L. Hopkins, T. Basil Smith III, and J.H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings pf the IEEE* **66, 10** pp. 1221-1239 (October 1978).

[10] Leslie Lamport, "Time, Clocks, and the Ordering of Events in Distributed System," *Communications of the ACM* **Vol. 21, No. 7** pp. 558-564 (July 1978).

[11] L. Lamport, Robert Shostak, and Marshall Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, pp. 382-401 (July 1982).

[12] Barbara Liskov and Alan Snyder, "Exception Handling in CLU," *IEEE Transactions on Software Engineering*, (November 1979).

[13] J.H. Wensley and et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of IEEE* **66, 10** pp. 1240-1255 (October 1978).

Constant Expected Time Randomized Byzantine Agreement Protocol without Shared Secrets and Cryptography

Sanjay Ranka, Anand Tripathi and Shiva Azadegan

Department of Computer Sciences

University of Minnesota,Minneapolis

Abstract

Reaching agreement in a distributed system with malfunctioning components is an important issue in building reliable computer systems. Byzantine agreement in a distributed environment has been an important problem in this context. Randomized algorithms for reaching Byzantine Agreement were proposed by Rabin[2] and further studied by many other researchers. These algorithms use shared secrets and cryptography to reach agreement in a constant expected number of phases. These shared secrets have to be boot-loaded during system intitation. This may become a serious drawback in practical situations. In this paper we present an algorithm which reaches agreement within a constant expected number of phases, independent of the number of processes in the system, without using shared secrets and digital signatures. The only shared information among processes is the knowledge of a logical configuration of all processes in a virtual ring. This information is boot-loaded with every process during system initiation. This algorithm can overcome $\left\lfloor \frac{n-1}{3} \right\rfloor$ faults, where n is total number of processes. The algorithm presented here achieves Byzantine agreement on binary values. However, this algorithm can be easily converted into multi-valued Byzantine agreement using the techniques described by some other researchers[3, 1]

This work was supported by a grant from Rome Air Developemt Center under the Post-Doctoral Fellowship Program.

1.O INTRODUCTION

We consider a set of n communicating processes $P = \{P_1, P_2, \ldots P_n\}$ with initial value $\{M_1, M_2, \ldots M_n\}$. The processes execute an agreement algorithm to agree on some common value m from the set $\{M_1, M_2, \ldots M_n\}$. The processes are assumed to be connected by a complete graph with n vertices i.e there is a communication link between any two processes. The system contains *proper* processes which follow the algorithm faithfully and *faulty* processes which may deviate sometimes maliciously, to prevent an agreement. We say that the system is proper if all proper processes have the same value for variable M. Byzantine agreement is defined as follows:

(1) If the system is proper in its initial state with m as the initial value of the proper processes,, then the algorithm terminates with system in the proper state with value m.

(2) If the system is not proper in the initial state, then the algorithm termiantes with system in a proper state with any value.

The Byzantine agreement has been an extensively studied algorithm for the past few years. It has been proved that any deterministic Byzantine agreement cannot be reached with less than $(t+1)$ phases, where t is the number of faulty processes in the system [4,3]. This lead to randomized algorithms [6] for reaching Byzantine agreement, which reach agreement in a small constant expected number of phases. Rabin's algorithm can tolerate upto $\left\lfloor \frac{n-1}{4} \right\rfloor$ faulty processes in the synchronous case but it assumes that some messages may be required to be authenticated by digital signatures. Morever it assumes a presence of a stream of secretly shared[7] bits distributed by a non-faulty dealer at system initiation time, large enough to last during system's operational life. Other researchers [8,5] have imporoved Rabin's algorithm in terms of the number of faulty processes that can be tolerated in order to achieve agreement. However, all these algorithms still require shared secrets and authentication. This turns out to be a serious drawback of the above schemes. Other algorithms, which do not require authentication and shared secrets and reach Byzantine agreement within a constant amount of expected time, can survive only $\sqrt{n}$ faulty processes [2,1].

In this paper we propose an algorithm that can tolerate upto $\left\lfloor \frac{n-1}{3} \right\rfloor$ faulty processes in synchronous cases and does not use any shared sequence of bits or digital signatures for authenticating messages.

We organize the remainder of the paper in the following manner. In section 2, we briefly give our model asssumptions. In section 3, we describe our algorithm. In section 4, we prove that the expected number of phases it will take to reach an agreement is a constant, independent of the number of faulty processes.

2.0 MODEL ASSUMPTIONS

Let P= $\{P_1, P_2, \ldots, P_n\}$ proceseses participating in the protocol. Assume that they have binary values $\{M_1, M_2, \ldots, M_n\}$ before the execution of the protocol. These processes are interconnected by a totally connected network. We assume synchronous communication among processes which implies that there exists a upper bound on the message delays between any pair of proper processes. We define a phase to be the interval of time in which each proper process is able to communicate with all other proper processes. A round consists of 2 phases of information exchange. One of the processes is designated to be the dealer for a particular round. This dealer is decided in a round robin fashion for each consecutive round i.e if P_1 is the dealer for the first round, then P_2 is the dealer in the second round and so on. For this purpose all processes form a virtual ring, and the knowledge of this virtual ring configuration is the only shared information which is required to be boot-loaded with every process during system initialization. Thus each process has the information about who is the dealer for the current round and who will be the dealer in the later round. The starting dealer for every agreement run is also chosen in a round robin fashion. Faulty processes may deviate from the algorithm and may maliciously, and possibly in collusion with each other, try to jeopardize the agreement.

Since the processes are interconnected by a totally connected network, each processor can recognize the sender of each message and faulty processes cannot change the message of a non-faulty processor. Each message is of the form (r,p,m) where r is the round number, p is the phase and m is a binary message.

3.0 THE ALGORITHM

A process P_p starts the algorithm by setting the value of variable x to its initial value M_p. This is followed by R rounds of message exchange, each having 2 phases. The number of rounds R is selected on the

```
Process P
x:= M_p
for Round :=1 to R do
begin
Phase 1: Send (Round,1,x) to all processes including itself
        If (Round,1,x) recieved from more than ⌊(n+t)/2⌋ processes
          begin
              Decided := true ; x:=m
          end
        else Decided:=false;
Phase 2: If p=dealer_for_this_round then
          begin
              If Decided then
                    send (Round,2,x)
              else
                    begin
                          x = random(0,1); {Function random returns either
                      0 or 1 with probability 1/2}
                          send(Round,2,x) to all other processes
                    end
          end
        else
      begin
              If recieved (r,2,m) from the dealer_for_this_round then
                    If not decided then x:=m
      end
end;
```

Algorithm: Randomized Byzantine Agreement

---

basis of desired level of confidence in the final agreement value as a correct agreement.

In phase 1, process sends its value x to all the processes and waits to recieve messages from every other process. If a message is not recieved within a particular amount of time, maximum time of the phase, it is assumed to be 0. A proper process is supposed to send a response within the maximum time of the phase. A process decides on a particular value of m if $\left\lceil \frac{n+t}{2} \right\rceil$ messages are of the same type and changes it's value of x to m. Otherwise its state is undecided.

In phase 2, if the process is a dealer and if it is decided it sends its value of x to everyone else. If it is dealer and undecided it randomly chooses between (0,1) with probability 1/2 and sends this value to every-one and updates its own value. If the process is not a dealer and undecided for that round, it changes its

value to the value recieved from the dealer. Otherwise it does not do anything.

The total number of messages sent by all the processes to each other in one round is n^2.

Theorem 1: An asynchronous system with n>3t and binary initial values executing the above algorithm will satisfy the following

1. At the end of first phase of every round two proper processes cannot decide on 2 different values.

2. If the system is proper with initial value m, then every proper process terminates the algorithm with x=m.

3. If the system is not proper, then with the probability of at least $1-\left[\frac{1}{2}\right]^{k(n-t)+l}\left[\frac{2l}{n}\right]^{t}$ every proper process terminates the algorithm after R rounds, where k=R div n and l=R mod n.

Proof

(1): Suppose P_1 and P_2 decide on X_1 and X_2 respectively at end of phase 1 of any round. Then P_1 must have recieved message X_1 from more than $\frac{n-t}{2}$ proper processes. Similiarly P_2 must have recieved message X_2 from more than $\frac{n-t}{2}$ proper processes. Thus at least one proper process must have sent message X_1 to P_1 and X_2 to P_2. Contradiction.

(2): Suppose if all the proper processes start with the same values of x at the beginning of any round. Let the value be m. We will show that all the proper processes will have the same values of x, equal to m, at the end of the round. Consequently, once an agreement is reached by all proper processes, this agreement will hold after each subsequent round. Hence if the system properly commences with value m, all proper processes will agree on the same value after every subsequent round. The following is the proof.

After the end of the first phase all the processes will recieve (n-t) messages of the same value. Since (n-t) > $\frac{n+t}{2}$ as $n > 3t$. Therefore all processes will decide, on the same value. In the second phase the decided processes do not change there values, independent of the dealer's value.

(3): Let us consider the scenario after completion of the first phase of any round before which the system was not in a proper state. The proper processes can be classified into two sets D and N referring to decided

and undecided processes respectively. By end of the first phase all the decided processes must have agreed on one particular value say m. There are 3 cases

1. Dealer is proper and undecided

2. Dealer is proper and decided

3. Dealer is faulty

In Case 1, the dealer will generate a random m =(0,1) each with probability $\frac{1}{2}$. This is the value which will be assumed by all undecided processes at the end of this round. If this value matches with the value of the decided processes with a probability $\frac{1}{2}$, all proper processes will have the same value and their will be an agreement at the end of this round.

In Case 2, the dealer will send its decided value to everyone and this value will be assumed by all undecided proper processes. Thus at the end of this round there will be an agreement.

Thus we conclude that there is a probability of at most $\frac{1}{2}$ of not reaching an agreement at the end of this round if the dealer is not faulty. Let us consider a sequence of y dealers ($y \leq n$).

$$D_1, D_2, \ldots D_y$$

The probability of x dealers being faulty out of these y dealers is

$$g(x) = \frac{\binom{t}{x}\binom{n-t}{y-x}}{\binom{n}{y}}$$

$$g(x) \leq \binom{t}{x}\left(\frac{y}{n}\right)^x$$

If there are x faulty processes out of these y processes the probability of not reaching an agreement after y rounds is at most $\left[\frac{1}{2}\right]^{y-x}$. Therefore, the probability of not reaching agreement in y rounds is at most

$$f(y) = \sum_{x=0}^{Min(y,t)} \binom{t}{x}\left[\frac{y}{n}\right]^x\left[\frac{1}{2}\right]^{y-x}$$

$$f(y) \leq \left[\frac{1}{2}\right]^y\left[1+\frac{2y}{n}\right]^t$$

Thus our claim is true for $y \leq n$. Let us consider a sequence of n dealers

$$D_1, D_2, \ldots D_n$$

The sequence will contain at least n-t proper processes, as the dealers are chosen in a round robin fashion. Thus probability of not reaching an agreement at the end of n rounds will be at most $\left[\frac{1}{2}\right]^{n-t}$ times the probability of disagreement at the beginning of the sequence. Thus for $y=kn+l$ it can be easily shown that the probability of not reaching an agreement is at most

$$\left[\frac{1}{2}\right]^{k(n-t)}\left[\frac{1}{2}\right]^{l}\left[1+\frac{2l}{n}\right]^{t}$$

Thus the probability of reaching an agreement is at least

$$1-\left[\frac{1}{2}\right]^{k(n-t)}\left[\frac{1}{2}\right]^{l}\left[1+\frac{2l}{n}\right]^{t}$$

□

4.0 ANALYSIS

Let the expected number of round for reaching agreement be E. Let

$$f(y)=\left[\frac{1}{2}\right]^{k(n-t)}\left[\frac{1}{2}\right]^{l}\left[1+\frac{2l}{n}\right]^{t}$$

where k= y div n, l = y mod n

Let P(y) be the maximum probability of achieving agreement in y^{th} round.

$$P(y)=((1-f(y))-(1-f(y-1))$$

$$=f(y-1)-f(y)$$

$$E\le\sum_{y=0}^{\infty}yP(y)$$

$$=\sum_{y=0}^{\infty}f(y)$$

$$=1+Z+\left[\frac{1}{2}\right]^{n-t}Z+\left[\frac{1}{2}\right]^{2(n-t)}Z\ldots$$

where $Z=\sum_{y=1}^{n}f(y)$

$$E\le 1+\frac{Z}{1-\left[\frac{1}{2}\right]^{n-t}} \le 1+\frac{8}{7}Z \quad \text{as } N>3t \text{ and } t>0$$

$$\frac{f(y+1)}{f(y)}=\frac{1}{2}\left[\frac{n+2y+2}{n+2y}\right]^{t} \quad \text{for } y<n$$

$$\leq \frac{1}{2}\left[1+\frac{2}{n}\right]^t$$

$$Z \leq \rho + \rho^2 + \ldots \rho^n, \textit{ where } \rho = \frac{1}{2}\left[1+\frac{2}{n}\right]^t$$

Since $n > 3t$, $\rho \leq \frac{1}{2}e^{\left[\frac{2}{3}\right]}$. Hence $\rho \leq 0.975$.

Now it can be easily shown that $Z \leq 39$ *and* $E \leq 46$. For $n > 5t$ the values are 3 and 4 respectively. Note that these are not the tightest bounds on E. With some more calculations, the value of E can be bounded to 39 and 3 respectively.

The following theorem states that for the case n>5t it is possible for a process to detect agreement when the system has indeed entered a proper state.

Theorem 2 a) For n>5t, if in round k some process, P_i, finds (n-t) messages with the same value, then by round $k+1$ all the processes in the system will be able to detect that agreement has been reached.

b) If the system is in agreement, then every process will find (n-t) messages with the agreement value.

(The proof is omitted.)

5.0 CONCLUSION

A new algorithm for achieving fast Randomized Byzantine Agreement has been presented in this paper. This algorithm does not require any secret sharing of information and digital signatures to authenticate. This algorithm can overcome upto $\left\lfloor \frac{n-1}{3} \right\rfloor$ malicious processors and reaches agreement in constant expected number of rounds. The number of messages required for each round are $o(n^2)$. The algorithm presented here achieves Byzantine agreement for binary values; this algorithm can be converted to a multi-value Byzantine agreement using the techniques described by some other researchers[9,5].

This algorithm, without having the practical drawbacks of earlier algorithms, takes a constant expected number of rounds to achieve agreement. In comparision to algorithms which do not use shared secrets and cryptography, it has a much better bound on the expected number of rounds required to achieve agreement.

REFERENCES

[1] Michael Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proceedings of the Second Symposium on Principles of Distributed Computing*, pp. 27-30 (1983).

[2] Gabriel Bracha, "An O(lg n) Expected Rounds Randomized Byzantine Generals Protocol," *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pp. 316-326 (1985).

[3] D. Dolev and H.R. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM Journal of Computing* **Vol. 12, No. 4** pp. 656-666 (November 1983).

[4] Nancy A. Lynch and Michael Fischer, "A Lower Bound for Time to Assure Interactive Consistency," *Information Processing Letters*, pp. 183-186 North-Holland, (June 1982).

[5] Kenneth J. Perry, "Randomized Byzantine Agreement," *IEEE Transactions on Software Engineering* **Vol. SE-11, No.6** pp. 539-546 (June 1985).

[6] Michael Rabin, "Randomized Byzantine Generals," *Proceedings of the 24th Symposium on Foundations of Computer Science*, pp. 403-409 (November, 1983).

[7] Adi Shamir, "How to Share a Secret," *Communications of the ACM* **Vol. 22, No. 11** pp. 612-613 (November 1979).

[8] Sam Toueg, "Randomized Byzantine Agreements," *Proceedings of the Third Symposium on Principles of Distributed Computing*, pp. 163-178 (August 1984).

[9] R. Turpin and B. Coan, "Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement," *Information Processing Letters* **Vol. 18** pp. 73-76 North-Holland, (February 1984).

A Protocol for Self-stabilization in Binary Trees

Saikeung Dong and Anand Tripathi
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Keywords: Multiprocessing, Process Synchronization, Distributed Control, Error Recovery

1. Introduction

A *self-stabilizing* system has the property of recovering from erroneous (*illegitimate*) states by continuing to execute its actions as in normal states. No additional mechanisms, other than those for normal operations, are required to recover from errors and enter a *legitimate* state. Such systems were first introduced by Dijkstra [1] and were called self-stabilizing systems. In this paper, we present a protocol for a self-stabilizing system consisting of a network of processes in the form of a binary tree and prove its correctness.

In his paper [1], Dijkstra gives three protocols that achieve self-stabilization, two of which are for processes connected in a ring and one for processes connected in a chain. In his design, Dijkstra defines legitimate states as those states in which exactly one privilege is present in the system. Our design is based on Dijkstra's model of self-stabilizing systems. The system consists of a network of processes, each of which can obtain the states of some but not necessarily all of the processes in the system. A process obtains the state of another process by reading the contents of the local memory of that process. Each process decides whether it has the *privilege* to make a *move* (i.e. local state transition); a privilege is defined as a Boolean function of its local state and the states of some other processes in the network.

An action by a process consists of checking whether it has one or more privileges present, and if so, then selecting one of the privileges and executing the corresponding move. To ensure that the privilege corresponding to a move holds when that move is executed, each such action is assumed to be executed

This work was supported by a grant from Rome Air Development Center under the Post-Doctoral Fellowship Program (Grant No. F-30602-81-C-0205).

atomically. Alternatively, this consistency can be ensured by a "daemon" [1] which selects privileges to be executed sequentially.

In a self-stabilizing system, with these local moves made by the individual processes, the whole system would eventually reach some globally consistent state (legitimate state) regardless of the initial system state. In addition, if by accident the system is in some illegitimate state, the local moves made by the processes should bring the system back to some legitimate state after a finite number of moves.

2. The protocol

The system of processes are connected in a binary tree-like topology, and our protocol is adapted from Dijkstra's second protocol [1]. For the sake of clarity in presenting the protocol, a process in the binary tree is assumed to have either no children or exactly two sons. The protocol can be easily extended to eliminate this restriction. We assume that each process knows if it is the left or right son of its father. A left son does not consult its right sibling in making its move while a right son needs the state information of its left sibling to decide if it has the privilege. Only adjacent processes can examine each other's local memory in order to obtain the state information of its neighbor. When a process is examining the states of its neighbor, the neighbor is not allowed to make a move.

Each process has two boolean state variables : UP and S. For any process, the state variables of its father, left son, right son, left sibling (if those nodes exist) are denoted by UP_F, S_F; UP_L, S_L; UP_R, S_R and UP_{LS}, S_{LS} respectively. By definition, the root has its $UP = false$ and a leaf has its $UP = true$. To simplify the protocol, a process that is a left child will have its $UP_{LS} = true$ and $S_{LS} = S_F$ although there are actually no siblings to its left.

For the root, its move is given by

if UP_L and UP_R and $(S = S_L)$ and $(S = S_R)$
then $S := \neg S$;

For an internal node other than the root, its move is given by

if $\neg UP$ and UP_L and UP_R and $(S = S_L)$ and $(S = S_R)$
then $UP := true$;

if UP and $\neg UP_F$ and UP_{LS} and $(S_F = S_{LS})$ and $(S \neq S_F)$
then begin $UP := false$; $S := S_F$; **end;**

For a leaf, its move is given by

if $\neg UP_F$ and UP_{LS} and $(S_F = S_{LS})$ and $(S \neq S_F)$
then $S := S_F$;

Observe that whenever a process u has its $UP = false$, there always exists at least one privilege in the subtree rooted at u. (This can be easily proved by induction on the size of the subtree rooted at u.) Now, $UP_{root} = false$ by definition. Hence, there is always at least one privilege in the system. In addition, if at some point all nodes except the root have their $UP = true$ and $S = S_{root}$ at the same instant, the system is stabilized and will remain stabilized if the protocol is observed. In this case, when the root makes a move, the privilege is passed to its left and then to its right subtree before the root regains the privilege. When a node $u \neq$ root obtains the privilege from its father, u makes a move, passes the privilege to its left and then to its right subtree before it regains the privilege from its sons and passes it up again.

Note that in this protocol we do not need any additional mechanisms such as a "daemon" or locking protocols to ensure the consistency of the actions. This can be seen from the fact that when a process has a privilege present based on the states of some other processes, then those processes cannot be possessing any privileges.

3. Correctness proof of the protocol

Under normal situation (i.e. when the system is stabilized), each of the processes in the binary tree takes turn to make a move in the order as described in the previous section. When the system has more than one privileges, each move of the root causes a "frontier" of stabilization to expand and diffuse from the root. Eventually this frontier covers all the leaves and the system is stabilized. The proof proceeds as follows :

Let G = G(V,E) be a binary tree with at least three nodes representing the network of processes.

Definition Let P = $<v_0, v_1, ..., v_k>$ be a path in G. Then P is an *up−path* if v_0 is the root and $\forall\ v_i$, $1 \leq i \leq k$, $UP_i = true$ and $S_i = \neg S_0$.

Definition An *up−tree*, T, is a subgraph obtained from a snapshot of G taken right after a move by the root which satisfies the following properties :

(i) root $\in$ V(T).
(ii) $u \in$ V(T) only if there exists an up-path to u in that snapshot.

Lemma 1 If a father node and a son node have been in the same up-tree once and since then they make their moves according to the protocol, then from that time onwards, $UP_{father} = true$ implies $UP_{son} = true$.

Proof Consider the first move (made by one of these two nodes) that results in a state contrary to that stated in the lemma. Before this move, $UP_{father} = UP_{son}$, no moves that are allowed in the protocol could have resulted in the aforementioned state. □

Theorem 1 For any two successive moves by the root, if the up-tree, T_1, corresponding to the first move has V(T_1) $\neq$ V(G), then the up-tree, T_2, corresponding to the second move, will satisfy V(T_1) $\subset$ V(T_2) and $|$ V(T_1)$|$ $<$ $|$ V(T_2)$|$.

Proof Let v be a node with the shortest distance from the root in G such that $v \in$ V(T_1) but $v \notin$ V(T_2). Let v 's father be u. [Observe that $v \neq$ root since by definition the root is in all up-trees. Also, $u \neq$ root otherwise by the fact that the root has the privilege to make a second move, v will be in V(T_2). Finally, $v \in$ V(T_1) implies $u \in$ V(T_1).] Let $S_{root} = x$ in T_1. In T_2, $S_{root} = \neg x$ since successive moves made by the root always flip S_{root}. v not in V(T_2) implies $UP_v = false$ or $S_v = S_{root}$ ($= \neg x$) in T_2.

Case 1. $UP_v = false$. By the choice of v, $u \in$ V(T_2). Thus, $UP_u = true$ in T_2. By lemma 1, $UP_v = true$. This is a contradiction.

Case 2. $S_v = \neg x$. Let P be the up-path to v in T_1. In T_1, $S_v = \neg x$. This implies v has made an even number of moves made possible by an even number of privileges obtained from above. The number of moves must be zero for otherwise v would have at least executed two privileges from u before the snapshot of T_2 is taken; and since $UP_u = UP_v$ in T_1, this implies u has at least executed two privileges from its father. Continuing this argument, one of the sons of the root in P would have executed at least two privileges obtained from the root after T_1 but before T_2 is formed. This is impossible. Hence, v has executed zero privileges from u when T_2 is formed. However, u has executed its privilege from above (as witnessed by $S_u = x$). At the time $S_u = x$, $UP_u = false$; UP_u cannot turn back to *true* without v 's making its

move. This contradicts the fact that $UP_u = true$ as implied by $u \in V(T_2)$.

To show $|V(T_1)| < |V(T_2)|$. Since $V(T_1) \neq V(G)$, there exists a node w in T_1 such that at least one of its two sons is not in T_1. For the root to execute its move again after the first move, S_w must be flipped. When it is flipped, $UP_w = false$. The only way to make $UP_w = true$ again is to have both sons of w having their $UP = true$ and $S = S_{root}$. The sons will hold their states until T_2 is formed. Thus, $|V(T_1)| < |V(T_2)|$. □

Lemma 2 Any node $u \neq$ root with $UP_u = false$ will eventually have $UP_u = true$.

Proof By induction on the size of the subtree rooted at u.

For the base, consider a subtree rooted at u with two leaves of G as its sons. Since UP_L and UP_R are *true* by definition, it suffices to show that S_L and S_R will equal S_u. (u will then have the privilege to move, setting $UP_u = true$.) This follows from the protocol for the leaves.

Now, consider a subtree rooted at u with size larger than three. Let $S_u = x$. Consider the left subtree of u rooted at v. (The right subtree can be handled similarly.)

Case 1. $S_v = x$ and $UP_v = true$. v holds its states until u makes a move by which $UP_u = true$.

Case 2. $S_v = x$ and $UP_v = false$. By induction hypothesis, UP_v will eventually be *true*. Observe from the protocol that this move made by v does not affect S_v. Thus $S_v = x$ and $UP_v = true$ and this becomes case 1.

Case 3. $S_v = \neg x$ and $UP_v = true$. v has the privilege to move, setting S_v to x and UP_v to false. Apply case 2 's argument.

Case 4. $S_v = \neg x$ and $UP_v = false$. By induction hypothesis, $UP_v = true$ eventually without affecting S_v. Apply case 3 's argument.

With both sons of u having $UP = true$ and $S = S_u$, u will make a move, setting UP_u to *true*.

□

Lemma 3 For any node $u \neq$ root, if $UP_u = true$ and there are no privileges from its father, the subtree rooted at u will eventually have no privileges.

Proof By induction on the size of the subtree rooted at u.

Consider a subtree, T, rooted at u with two leaves of G as its sons. Since UP_u, UP_L, UP_R are

true, and given no privileges from u 's father, T has no privileges. This establishes the base of the induction.

Now, consider a subtree rooted at u with size larger than three. Consider the left subtree of u rooted at v.

Case 1. $UP_v = true$. v has no privileges from u. By induction hypothesis, the left subtree of u will eventually have no privileges.

Case 2. $UP_v = false$. By lemma 2, UP_v will become *true*. Apply case 1 's argument.

Similarly, the right subtree of u will eventually have no privileges. Thus the subtree rooted at u eventually has no privileges. □

Theorem 2 Within finite number of moves, G is stabilized.

Proof If the root executes its privileges infinite number of times, the successive up-trees formed will eventually covers G by theorem 1. At that point, G is stabilized. Otherwise, the root executes a finite number of moves. We will show that this cannot happen. Consider the last move made by the root. Let $S_{root} = x$ after this move. The left son, u, of the root will make a move, setting $UP_u = false$ and $S_u = x$. By lemma 2, UP_u will eventually have value *true*. And since the root has made its last move, u cannot obtain another privilege from the root. By lemma 3, the subtree rooted at u eventually has no privileges. Similarly, the right subtree of the root will eventually have no privileges. At this point, only the root has the privilege to move, and it will make a move provided it is not infinitely lazy. This contradicts the finite number of moves made by the root. □

References

[1] E. W. Dijkstra, "Self-stabilization in spite of distributed control", *Communications of the ACM.*, (Nov., 1974) 17:643-644.

[2] E. W. Dijkstra, "A belated proof of self-stabilization", *Distributed Computing*, (1986) 1:5-6

An Improved Algorithm for Termination Detection in Asynchronous Distributed Computations

Anand Tripathi and Saikeung Dong
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Keywords: Distributed Computing, Operating Systems, Networks, Process Control

1. Introduction

Topor [4] gives an elegant derivation of a termination detection algorithm, however, his algorithm implicitly requires that message delays be bounded. Otherwise, the algorithm may declare termination (of a distributed computation) while some processes are still actively engaged in the computation. We give an upper bound on the message delays that is necessary and sufficient for Topor's algorithm to function correctly in a system of processes where message transmission is not instantaneous as in CSP [2]. Next we give an improved (termination detection) algorithm that allows arbitrary (but finite) message delays in a system. Finally, we prove the correctness of the improved algorithm.

Our algorithm is similar to that of Misra [3] in the sense that we use special messages on every communication channel to flush out messages in transit. However, our algorithm uses a spannng tree instead of a cycle (consisting of all channels in a system) as is used in [3]. We assume the readers are familiar with the work of Dijkstra [1] and Topor [4]. (See appendix for a brief description of Topor's algorithm.)

2. Preliminaries

We assume there is a set of processes cooperating in some computation. The processes are modeled by a graph $G = (V,E)$ where V represents the set of processes and E the communication channels among the processes. Processes can be *active* or *idle*. Only an active process may send messages to its neighbors in G. An active process may turn from active to idle at any time but an idle process can only become active on the receipt of some message. All messages are received in the same order as they are sent; this applies to

This work was supported by a grant from Rome Air Development Center under the Post-Doctoral Fellowship Program (Grant No. F-30602-81-C-0205).

all types of messages. A distributed computation is said to have terminated if all processes are idle and there are no messages in transit.

To establish an upper bound on message delays for Topor's algorithm to function correctly, we need the following notations. Let :

T be a fixed spanning tree in G. (The tree used in Topor's algorithm.)

For any node u, v in V,
l_u is the distance (level) of u from the *root* in T;
T_u is the subtree of T which is rooted at u;
h_u is the height of T_u where $h_u = \max \{ l_w - l_u : w \in V(T_u) \}$.

If u sends a message to v, $delay(u,v)$ is the time between u's sending and v's receipt of the message.

δ (> 0) is the minimum message delay time for all $e \in E$. Thus, $\delta \leq delay(u,v)$.

$[a,b]$ with $a \leq b$ is the time interval between global time a and global time b, including both a and b in the interval. Similarly, $[a,b)$ is the interval between a and b, including a but excluding b. $(a,b]$, (a, b) are defined analogously.

R_i ($i \geq 1$) is the global time when the *root* sends the i^{th} wave of (repeat) signals in Topor's algorithm or when it declares termination after $(i - 1)$ waves of signals. By definition, $R_0 = 0$.

τ_{u_i} is the global time when it is the i^{th} time u sends a token up to its father.

Note : For $i \geq 1$, $\tau_{u_i} < R_i$, $R_i - \tau_{u_i} \geq l_u \delta$, $R_i < \tau_{u_{i+1}}$ and $\tau_{u_{i+1}} - R_i \geq (l_u + 2h_u)\delta$.
The last inequality comes from the fact that u cannot send its token up until it has received a signal from above (which takes at least $l_u \delta$), passed signals to nodes below (which requires at least $h_u \delta$) and got new tokens from nodes below (which takes at least $h_u \delta$).

If before the distributed computation has started, all nodes are white with leaf nodes having white tokens and there are no messages in transit, it is possible for Topor's algorithm to declare termination without the sending of any (repeat) signals by the *root*. This can happen only if no nodes have sent any messages throughout the computation. In reality, this is unlikely to be the case; some messages for computational purposes are exchanged before termination, hence we assume without loss of generality that the *root* will send at least one wave of signals downwards. In the following discussion, we exclude the trivial case of no signals being sent by the *root* and assume that the *root* will not declare termination at R_1, i.e. it sends at least one wave of signals.

3. Analysis of the bound on message delays for Topor's algorithm

Lemma 1 For Topor's algorithm to function correctly in a system of asynchronous communicating processes, it is necessary that for any distinct $u, v \in V$, $delay(u,v) \leq (l_u + l_v + 2h_v)\delta$.

Proof Suppose not, then the following scenario can occur. Let $p, q \in V$ be two sons of the *root* in T. Pick $u \in V(T_p)$ and $v \in V(T_q)$ with $(u,v) \in E$. u sends a message to v and turns black. u then immediately sends a black token to its father and paints itself white. Meanwhile, all other nodes are white and idle and the *root* has received white tokens from every child except p. Say the black token sent by u takes exactly $l_u\delta$ time to get to the *root*. After that, the *root* initiates a new wave of signals since having received a black token forbade it to declare termination. If the signal reaches v in exactly $l_v\delta$ time and v receives all white tokens from its sons in $2h_v\delta$ time, then $(l_u + l_v + 2h_v)\delta$ time has elapsed since u's sending of message to v. By assumption, this message has not arrived at v yet. v as well as u can send white tokens up, causing the *root* to declare termination. However, v can become active after receiving the message in transit, invalidating *root*'s conclusion of the system. □

To prove the sufficiency of the above bound, we examine the time when the *root* initiates different waves of signals. If the *root* declares termination at time R_{i+1} ($i \geq 1$), we show that for any node u, all messages sent by u before time τ_{u_i} are all received by the receiving nodes and there are no messages sent by u after τ_{u_i}.

Lemma 2 For $i \geq 1$, if $delay(u,v) \leq (l_u + l_v + 2h_v)\delta$ and the *root* does not declare termination at R_i, then any messages sent before τ_{u_i} from u to v are received by $\tau_{v_{i+1}}$.

Proof The proof is by induction on i. For $i = 1$, the *root* does not declare termination at R_1 since we assume that the *root* sends at least one wave of signals. Hence τ_{v_2} exists. From the Note in the previous section, $\tau_{v_2} - \tau_{u_1} \geq (l_u + l_v + 2h_v)\delta$. By the fact that u sends the message to v before τ_{u_1} and the bound on $delay(u,v)$, v must have received the message by τ_{v_2}. Thus, the basis of induction is established.

For $i = k + 1$, assume the *root* does not declare termination at R_{k+1}. By induction hypothesis,

we only need to consider messages sent by u to v in the interval $[\tau_{u_i}, \tau_{u_{i+1}})$. Again $\tau_{v_{i+2}}$ exists and $\tau_{v_{i+2}} - \tau_{u_{i+1}} \geq (l_u + l_v + 2h_v)\delta$. Hence by time $\tau_{v_{i+2}}$, v should have received all messages sent by u during $[0, \tau_{u_{i+1}})$ because of the bound on $delay(u,v)$. □

Lemma 3 If the *root* declares termination at R_{i+1} ($i \geq 1$), then no node u has sent a message in the interval $[\tau_{u_i}, R_{i+1}]$ and u remains idle and white in $[\tau_{u_{i+1}}, R_{i+1}]$.

Proof Assume some nodes become active or black between the interval of their last sending of tokens up and the declaration of termination by the *root* at R_{i+1}. Pick from these nodes a node u such that it is the earliest (in global time after R_i) to be activated. Say u is activated by some node w. By lemma 2, w cannot have sent the wake-up message to u before τ_{w_i}. Also, w cannot have sent the message in $[\tau_{w_i}, \tau_{w_{i+1}})$ for otherwise the *root* cannot have declared termination. Thus w has turned from idle to active in $[\tau_{w_{i+1}}, R_{i+1})$ before it can send the message to u. This implies that w is activated at least δ time before u. This contradicts the choice of u. □

By ensuring that the *root* sends at least one wave of signals together with the bound on $delay(u,v)$, lemma 2 and 3 guarantee that when the *root* declares termination, there are no messages in transit and all nodes are inactive and white. Thus we have :

Theorem 1 Given that the *root* sends at least one wave of signals, Topor's algorithm will function correctly in a system where message transmission is not instantaneous if and only if for any two distinct nodes u, v, $delay(u,v) \leq (l_u + l_v + 2h_v)\delta$.

4. An improved termination detection algorithm

Given $G = (V,E)$, our algorithm will use a fixed spanning tree T. The *root* of T will be responsible for termination detection.

Definition For any node $u, v \in V$ such that $(u,v) \in E$,
(a) u is *above* v if $l_u < l_v$.
(b) u is *below* v if v is above u.
(c) u is a *sibling* of v if $l_u = l_v$.

Our algorithm is a modification of Topor's algorithm. We shall use four colors to color the nodes : blue, grey, black and white. Roughly speaking, when the algorithm has been initiated, blue and grey nodes are those that have seen a new wave of signals but have not sent their tokens up while black and white nodes are those that have sent their tokens up. There are also two colors for the token : black and white. A black token means there are possibly messages in transit and it is not safe to declare termination while a white token represents a possibility of no messages in transit from a local point of view of some node. Our algorithm consists mainly of alternately sending a wave of signals from the *root* downwards, propagating to the leaves and a wave of tokens upwards in the reverse direction, changing the color of nodes if necessary. If the *root* has received white tokens from all nodes below and is idle and blue, then it will declare termination.

Initially, i.e. before the computation has started, all nodes are white, there are no nodes with tokens or signals and there are no messages in transit. The algorithm is informally given as a set of rules.

For initiating the algorithm :

Rule 1 When the *root* is idle, it sends signals on all edges to nodes below and paints itself blue. (This rule is applied only once.)

On receiving a token :

Rule 2 A node on receiving a black token from below paints itself grey. Otherwise, there are no changes to its color.

For sending tokens upwards :

Rule 3 A node u other than the *root* (u may be a leaf node) sends tokens to all nodes above when :
(a) it is idle and
(b) it has received tokens from all nodes below it (this is trivially satisfied for a leaf node) and
(c) it has received signals from all its siblings.
The color of u and the color of the tokens u sends are as follows :
If u is blue, it sends out white tokens and paints itself white.[1]
If u is grey, it sends out black tokens and paints itself black.
After sending out tokens, u loses its tokens.

For normal signal sending :

Rule 4 After the *root* has received all tokens from below and is idle, if it is blue, it declares termination.[2] Otherwise, it loses all its tokens, sends signals to nodes below and paints itself blue.

---

[1] In this case, all tokens received by u must be white by Rule 2.

[2] In this case, all tokens received by the *root* must be white by Rule 2.

Rule 5 When an internal node other than the *root* has received signals from all nodes above, it waits until it is idle, sends signals to its siblings and all nodes below and paints itself blue.

Rule 6 A leaf, on receiving signals from all nodes above, waits until it is idle, sends signals to its siblings and paints itself blue.

For sending messages :

Rule 7 When any node $u \in V$ sends a message along any of its incident edge, it changes color as follows :
If u is white, it changes to black.
If u is blue, it changes to grey.
Otherwise, there are no changes to u's color.

5. Correctness Proof

We need to show that when the *root* declares termination, all nodes in V are white and idle. If this is the case, we claim there are no messages in transit. If there is a message sent by u in transit, either it is sent *before* the last time u turns blue in which case the message should have been flushed out by the signals (if the receiving node is a sibling of u or is below u) or tokens (if the receiving node is above u) u sent or the message is sent *after* u turns blue in which case u is black when the *root* declares termination and is impossible. Hence it suffices to establish for the above algorithm the following :

Theorem 2 When the *root* declares termination, all nodes in V are white and idle in the system.

Proof Suppose not, then there exists some node u such that its color is not white or is active. If u is blue or grey, then the *root* could not have declared termination for u has not sent out its tokens yet. If u is white and active, it must have received a message from a grey or black node w that wakes up u. If w is grey right after sending this message, the *root* could not have declared termination because w has seen the last wave of signals and will send black tokens up. Thus, w is black (right after sending this message). If w was grey before it turns black, again the *root* could not have declared termination. Hence, w must have changed from blue to white to black. Among all nodes that have changed from blue to white to black, consider the one node x that has the change from white to black the earliest since the last time the *root* sent signals. Since x had been white, all its neighbors have seen the last wave of signals from the *root*. All messages sent by x's neighbors to x before they turned blue the last time were received by x when x turned white. Hence the node y that wakes up x must have

sent the wake-up message to x after it turned blue but before it sent its token up (this follows from the choice of x), i.e. y changed its color from blue to grey to black. Again the *root* could not have declared termination. □

It can easily be seen that if the distributed computation has ended, the above algorithm will eventually declare termination.

References

[1] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations", *Information Processing Letters*, 16(5) (1983) 217-219.

[2] C.A.R. Hoare, "Communicating Sequential Processes", *Communications of the ACM.*, 21(8) (1978) 666-677.

[3] J. Misra, "Detecting termination of distributed computations using markers", *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, 290-294.

[4] R.W. Topor, "Termination detection for distributed computations", *Information Processing Letters*, 18(1) (1984) 33-36.

Appendix : Topor's Algorithm

Given a graph $G = (V,E)$, this algorithm will make use of a fixed spanning tree T in G. Initially, each leaf has a token.

Rule 0 An idle leaf that has a token transmits a token to its parent; an idle internal node that has received a token from each of its children transmits a token to its parent; an active node does not transmit a token. When a node transmits a token, it is left without any tokens.

Rule 1 A node sending a message becomes black.

Rule 2 A node that is black or has a black token transmits a black token, otherwise it transmits a white token.

Rule 3 A node transmitting a token becomes white.

Rule 4 If *root* has received a token from each of its children, and it is active or black or has a black token, it becomes white, loses its tokens, and sends a repeat signal to each of its children. If *root* is white, idle and tokens received from all its children are white, it declares termination.

Rule 5 An internal node receiving a repeat signal transmits the signal to each of its children.

Rule 6 A leaf receiving a repeat signal is given a white token.

Fault Diagnosis in Distributed Systems

Vijay Raghavan
Saikeung Dong
Anand Tripathi

Computer Science Department
136 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

1. Introduction

With the increase in the number of processing elements, computational power and complexity of interconnection, the reliability and fault tolerance of distributed systems have become areas of acute concern. A distributed system, like any other digital system, is subject to faults which make it deviate from its specified behavior; unlike other systems, however, most distributed systems come with a promise to deliver reliability and high availability of resources even in pathological cases of processor failures. Indeed, this is one of the main reasons for having distributed systems at all. To support the claims of fault-tolerance and reliability, such systems need to have the ability to diagnose faults when they occur and initiate recovery procedures.

A self-diagnosing system is one in which faults can be isolated to within replaceable parts of the system. As noted in [Kime80] fault diagnosis in distributed systems is generally a 2-step procedure- it involves both the *detection* of a fault when it occurs and its *location*, before repair or graceful degradation may be initiated. In this report, our principal concern is with the latter. We refine the notion of fault location in later sections, but it may be pointed out that the purpose of fault location is to identify a superset of the units which must definitely be faulty to cause the *syndrome* which afflicts the system.

Several models have been proposed for fault diagnosis in distributed systems. In this report, we restrict ourselves to just one model and its offshoots. We illustrate the salient features of this model with examples and explore the interesting theoretical contributions of this model. Finally, we identify a host of

unsolved problems with our own preliminary results and map the course of future research to be done in this area.

2. The PMC model

In the two decades since the introduction of the so-called PMC model [Prep67] , significant progress has been made in the development of the theory associated with the model. The reasons for this are not difficult to find: first, the simplicity and elegance of this model have made it appealing to both theoretical scientists and system designers alike. Second, the universality of the model makes it suitable for capturing the essence of diagnosis in distributed systems. Third, the model permits a level of abstraction where faultiness of a processor/software module is reduced to being just 2-valued: faulty or non-faulty. While this may not be the best picture of "real-world" processors and modules, which are susceptible to a fuzzy behavior between the two extremes of being faulty and non-faulty, this simplification has the beneficial effect of enabling diagnosis algorithms developed for the PMC model to be applicable in all layers of the system architecture. Indeed, one may drop the distinctions between hardware processors and software modules and just talk of *units* when modeling the system for the purpose of fault diagnosis.

A distributed system is one in which computational tasks are performed by multiple units. There are 2 principal assumptions which are made in the PMC model about the distributed system being modeled: first, the variables which characterize the fault behavior of units - Mean Time Between Failure, Mean Time To Repair *inter alia*- are assumed to be *independent* random variables; second, the units are assumed to have the capability to administer tests among themselves for the purpose of diagnosis. We concern ourselves neither with the precise nature of these tests nor with how and when they are actually administered, beyond insisting that they be *complete*, i.e. a fault-free unit should always correctly identify the units it tests as being faulty or fault-free.

The system that is to be diagnosed is partitioned into logical units. These units need not be similar in their functionality within the distributed system, except in their ability to test singly or in combination, another one of the units. The outcome of the tests may be classified simply as "pass" or "fail", indicating that the testing unit evaluates the tested unit as being fault-free or faulty, respectively. We assume that the evaluation is meaningful only if the testing unit itself is fault-free, otherwise the outcome is unreliable. This

assumption is known as the *symmetric invalidation* assumption and forms the basis of many offshoots of the PMC model. An alternative assumption, which is also frequently used (and considered by some to be more representative of real systems and real tests) is the *asymmetric invalidation* assumption, wherein a "pass" outcome necessarily implies that the tested unit is fault free, but a "fail" outcome only implies that either the tested unit or the testing unit or both are faulty. The rationale behind this assumption [Bars76, Kime80] is that a complete test in systems composed of complex units entails the checking of a large number of responses from the tested system, rendering it extremely unlikely that the faults in the units performing the test would completely cancel the faults in the unit under test causing a test to pass, when it should have failed.

The test system is modeled as a directed graph $G(V,E)$, with the units represented by vertices in the graph and the tests by directed edges. Thus if $<i, j>$ is a directed edge from unit u_i to unit u_j, the unit u_i tests the unit u_j. This directed graph is called the *connection assignment* of the system.

Weights are assigned to edges depending on the test outcomes: the weight a_{ij} associated with the edge <i, j> is defined as follows:-

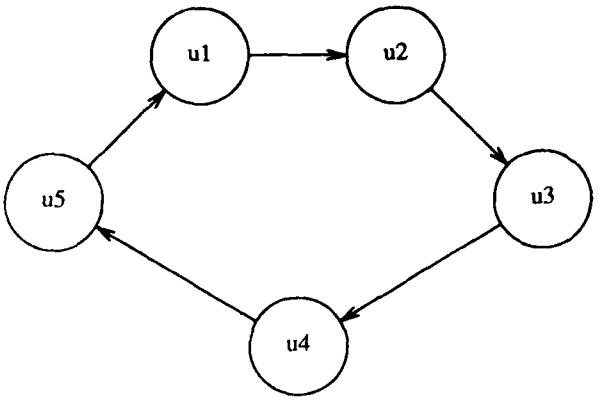
$$a_{ij} = \begin{cases} 0 & \text{if } u_i \text{ tests } u_j \text{ with outcome } pass \\ 1 & \text{if } u_i \text{ tests } u_j \text{ with outcome } fail \end{cases}$$

The set of test outcomes a_{ij} is called the *syndrome* of the system. There are $2^{|E|}$ possible syndromes for any connection assignment.

We use an example to motivate the discussion and definitions which follow.

Example 1

Consider a system composed of 5 units $u_1, u_2, \ldots u_5$ whose connection assignment is in the form of a ring. A syndrome for the system can be represented as a 5-bit vector $(a_{12}, a_{23}, a_{34}, a_{45}, a_{51})$.



Assume exactly one of the units, say u_1, is faulty. Then

$$a_{23} = a_{34} = a_{45} = 0; \quad a_{51} = 1$$

i.e. u_5 correctly identifies u_1 as being faulty, and

$$a_{12} = x \quad i.e.\ 0\ or\ 1$$

since u_1, being faulty, may or may not diagnose u_2 properly.

Thus the syndrome for exactly one of the units being faulty can be only of the form

$$(x\ 0\ 0\ 0\ 1)$$

or one of its cyclic permutations. Moreover, it can easily be proved that the connection assignment is capable of identifying the faulty unit when exactly one unit is faulty.□

The example raises the question of whether the modeled system is capable of identifying more than 1 fault. This question is ambiguous on 3 counts since the system's "capability to identify more than one fault" is open to 3 different interpretations:

First, it could mean the capability of locating upto t faults ($t > 1$) *instantly*, i.e. with just one syndrome. Note that an upper bound on the number of faults is necessary since the entire set of units, V, is always a consistent fault set for any possible fault set. Moreover, if a system has the ability to identify fault sets unequivocally, this necessarily implies that the number of faulty units is less than $\left\lfloor \frac{n-1}{2} \right\rfloor$.

Second, it could mean the capability to identify at least 1 fault if the number of faulty units do not exceed t. In this case, we would be able to identify the remaining faulty units after replacing the faulty unit with a non-faulty one and repeating the test. This might involve as many repetitions of the test as the size of the fault set, and therefore represents an approach which is a compromise between having a small number of test links and performing a small number of tests.

Third, the word "identification" could be interpreted as meaning "locating within a set which contains the units sought". Thus, we could be questioning the system's ability to isolate the set of faulty units to be within a larger set, for we could then replace the larger set with impunity, knowing that all the faulty units have been replaced, albeit with some fault-free units. This represents the system designer's willingness to sacrifice a few good units for the sake of getting a fast response time in the fault detection and location phase before initiating system repair or graceful degradation.

All three interpretations are equally valid and give rise to 3 different diagnosability measures of a system.

Definition 1

A system of n units is *one-step t-fault diagnosable* (or simply *t-diagnosable*) if all faulty units within the system can be identified without replacement provided the number of faulty units present does not exceed t.

Definition 2

A system of n units is *sequentially t-diagnosable* if at least one faulty unit can be identified without replacement provided the number of faulty units present does not exceed t.

Definition 3

A system of n units is k-step *t/s-diagnosable* if by no more than k applications of the diagnostic tests, a set of size not more than s units can be identified, such that all faulty units are within the set, provided the number of faulty units present does not exceed t.

If a system is 1-step t/s-diagnosable, we say simply that it is t/s-diagnosable.

Example 2

Consider the system of *Example* 1 again. The syndrome (x0001) is compatible with 2 possible fault sets: $\{u_1,u_2\}$ and $\{u_1\}$. Therefore the system is not 2-diagnosable. However, it *is* sequentially 2-diagnosable as we demonstrate below:

Since there are no more than 2 faults in the system, there must always be two fault-free processors adjacent to each other. So there is definitely some edge with weight 0, no matter what the syndrome is. Assume, without loss of generality, that $a_{0,1}$ is 0.

Case 1 Suppose that $a_{5,1}$ is the only link with a 1 weight. Then, if u_1 is assumed to be fault-free, we have to conclude that $u_2,u_3,\ldots,u_5$ are all fault-free. But then $a_{5,1} = 1$ implies that u_1 is faulty, which is a contradiction. Hence, in this case u_1 must be faulty.

Case 2 Suppose that $a_{5,1}$ is also a link with a 0 weight. Then if u_2 is assumed to be faulty, we have to conclude that both u_1 and u_5 are also faulty. But then we have more than 2 units which are faulty, which contradicts the assumption that there are at most 2 faulty units in the system. Hence, u_2 must be fault-free and we just have to follow the links after u_2 around the ring until we come to a link with a weight of 1. The unit that this link points

to must be faulty.

Case 3 Suppose that $a_{5,1}$ is not the only link with a 1 weight. Then consider each of the possible following sub-cases distinguished by the 5-bit vector $X = (a_{5,1}, a_{1,2}, \ldots, a_{4,5})$. In each of the sub-cases the assumption that not more than 2 processors are faulty immediately leads to the identification of a faulty processor.

$X = (10001)$: Then u_5 must be faulty.

$X = (10010)$: Then u_4 must be faulty.

$X = (10011)$: Then u_4 must be faulty.

$X = (10100)$: Then u_5 must be faulty.

$X = (10101)$: Then u_3 must be faulty.

$X = (10110)$: Then u_3 must be faulty.

$X = (10111)$: Then u_3 must be faulty.□

3. Main Results and Open Problems

There are three main kinds of problems in the distributed system diagnosis environment as envisaged by the PMC model, each of which is discussed below with a summary of the main results.

3.1. The Characterization problem

It is natural to ask what kind of system architecture or interconnection strategy is to be adopted to design a distributed system having a certain diagnosability. Since we have three different diagnosability measures, this question then is really 3 different characterization problems:

1.The t-characterization problem: Given a certain t, (i.e an upper bound on the number of faulty units) what are necessary and sufficient conditions for a distributed system to be t-diagnosable?

Note that this problem (and as a matter of fact, the other 2 problems below) can be answered in purely graph-theoretical terms, which is yet another vindication of the PMC model. A system designer can assure himself of a certain amount of diagnosability if a complete characterization is possible, for then he only has to ascertain that the connection assignment for his system satisfies the requirements of diagnosability.

The first complete t-characterization appeared in [Haki74] We give below a simpler version

([Sull84, Alla75]):

A directed graph $G(V,E)$ is t-diagnosable if and only if

$$\forall Z \subseteq V \quad Z \neq \emptyset \Rightarrow \frac{|Z|}{2} + |\Gamma^{-1}Z| > t.$$

An application of this theorem reveals the following facts about t-diagnosable systems: first, the number of units must be at least $2t+1$, else the system cannot be t-diagnosable; second each unit must be tested by at least t other units, else certain scenarios of t-fault situations will remain undiagnosable.

Designs for t-diagnosable systems which are symmetric (i.e the interconnection strategy is the same for every processor) and optimal (i.e using the minimum number of testing links) have been achieved in the so-called $D_{\delta,t}$ systems ([Prep67]).

2.The sequential t-characterization problem: Given a certain t, what are necessary and sufficient conditions for a distributed system to be sequentially t-diagnosable?

The first paper on the PMC model introduced the concept of sequential diagnosability and gave very weak necessary conditions. It also dealt with the special class of single-loop systems and gave a complete characterization for sequential diagnosability in such systems. (Interestingly enough, at the time the paper was published, the authors did not realize that their characterization of single loop systems was complete. In a separate paper [Prep68] , Preparata demonstrated that the "sufficient" conditions were, in fact, necessary also). Since then, the sequential t-characterization problem has been solved for other special classes of systems (see for example, [Karu79]); however, as far as can be determined, this problem still remains open for general connection assignments.

3.The t/s-characterization problem: Given a certain t and s, what are necessary and sufficient conditions for a distributed system to be t/s-diagnosable?

This is another problem for which there appears to be no published solution for arbitrary connection assignments. Again, characterization for special classes of systems have been achieved. For example, Karunanithi and Friedman have addressed single-loop systems in [Karu79] ; Chwa and Hakimi have characterized the so-called $D_{\chi,\delta,t}$ systems and the restricted class of systems which are

t/t-diagnosable ([Chwa81]); Sullivan in [Sull84] , has given (without a proof) a characterization of $t/t+1$-diagnosable systems (which because of a typographical error is actually incorrect as stated). It also appears that no significant work has been done in designing optimal or sub-optimal arbitrary t/s-diagnosable systems, although it must be mentioned again that for special classes of systems, this issue has been settled satisfactorily (see for example, [Chwa81]).

3.2. The diagnosability problem

The flip side of the characterization problems are the corresponding diagnosability problems, one for each diagnosability measure defined above. Instead of asking for the characterization of a distributed system which supports a certain diagnosability measure, here we assume that the *inter-connection strategy* for diagnosis (i.e the connection assignment) has already been given to us and want to determine the maximum diagnosability that it can support.

t-diagnosability: Allan et al. [Alla75] introduced the concept of the *diagnosability number*, which is the largest number of faults, t, that a system with a given connection assignment $G(V,E)$ can tolerate and still remain *t-diagnosable*. For about a decade or so, only algorithms with time complexities exponential in n, the number of units in the system, were known for determining the diagnosability number for arbitrary connection assignments; recently, Sullivan ([Sull84]) used network flow techniques to obtain a remarkable $O(n^{2.5})$ algorithm, thus scotching the suspicion that a polynomial time algorithm did not exist.

Sequential t-diagnosability: The corresponding algorithm for determining the sequential t-diagnosability number for arbitrary connection assignments has not appeared in the published literature so far. In fact, it is not even known whether this problem is tractable (in terms of polynomial time solvability) or not.

t/s-diagnosability: Sullivan, in his landmark paper [Sull84] on a polynomial time algorithm for t-diagnosability, also proved that t/s-diagnosability is co-NP complete, i.e the following decision problem

Given a directed graph G(V,E) and positive integers t and s, is G t/s-diagnosable?

cannot be answered in time polynomial in n, unless the old debate of whether P equals NP ([Gare79]) is settled affirmatively. Actually, negative results like this abound in the diagnosis area (see for example [Mahe76, Fuji78] and [Soma86]); nevertheless, the theoretical importance of such results cannot be gainsaid. Polynomial time algorithms have been achieved for special classes of t/s-diagnosable systems: in [Sull84] , for example, is an algorithm for t/t-diagnosable systems, which can be generalised to yield efficient algorithms for $t/t+k$-diagnosable systems as long as k is much smaller than t.

3.3. The diagnosis problem

Even when a system has a demonstrable amount of diagnosability, there still remains the practical problem of diagnosing faults when they occur. This is perhaps the area of utmost concern to the system designer and surprisingly, is the area where there are the fewest results. Again, we examine the main results in diagnosis algorithms under the 3 measures of diagnosability:

t-diagnosis: Given a directed graph G(V,E) which is t-diagnosable, and a syndrome in a t-fault situation, find the unique fault set consistent with the syndrome.

The above problem, which is as old as the PMC model itself, eluded all attempts at an efficient solution for 17 years. Kameda et al. [Kame75] , gave an $O(n^3)$ algorithm for it, but as pointed out in [Corl76] , the algorithm was flawed by a technical error, which nevertheless does not render it unusable ([Madd77]). Only very recently ([Dahb84]), did Dahbura and Masson come up with a really elegant $O(n^{2.5})$ algorithm which uses the concepts of maximal matching and minimum vertex covers in undirected graphs. As such, all 3 problems (characterization, diagnosability and diagnosis) for the t-diagnosability measure can be considered to be completely solved.

Sequential t-diagnosis: Given a directed graph G(V,E) which is sequentially t-diagnosable, and a syndrome in a t-fault situation, find at least one unit which is definitely faulty, regardless of which set of units, among all the possible consistent fault sets for the syndrome, actually caused the

syndrome.

As mentioned earlier, the characterization and diagnosability problems for this class of directed graphs are still unsolved in the literature. Unfortunately, the same is true of diagnosis. Except for the restricted classes of single-loop and $D_{\delta,t}$ systems ([Prep67] and [Karu79] respectively), no attempt has been made to solve the sequential t-diagnosis problem for systems with arbitrary connection assignments.

t/s-diagnosis: Given a directed graph which is t/s-diagnosable, and a syndrome in a t-fault situation, locate a set X of cardinality no greater than s, such that every consistent fault set for the syndrome is a subset of X.

Here again, no decent algorithm is known, although Yang et al. ([Yang86]) gave an $O(n^{2.5})$ algorithm for t/t-diagnosable systems, using ideas from [Dahb84] Also, Chwa and Hakimi ([Chwa81]) have given an optimal algorithm for $D_{X,\delta,t}$ systems (which are restricted versions of t/t-diagnosable systems). So, except for the co-NP completeness result of Sullivan, little is known about general t/s-diagnosable systems.

The following table summarizes the state of affairs in the fault-diagnosis area:

Diagnos-ability measure	**Problem type**		
	Characterization	Diagnosability	Diagnosis
t	Solved	Solved	Solved
Seq. t	Open; but solved for special graphs	Open; but solved for special graphs	Open; but solved for special graphs
t/s	Open; but solved for special graphs	Solved: generally intractable	Open; but solved for special graphs

4. Our results and directions for future research

We have used concepts from set partitioning to solve the characterization problems mentioned in the previous section and which are still considered open in the literature[Ragh86] The characterization reveals some very interesting and heretofore unknown properties of t/s-diagnosable systems. We hope to take advantage of these properties to generalise the earlier work ([Dahb84] and [Yang86]) to produce an efficient algorithm for $t/t+k$-diagnosable systems. We are in the process of writing up our results for publication in the *IEEE Transactions on Computers*.

The main thrust of our immediate research will be in solving the other open problems mentioned in the previous section. There are two other areas in fault diagnosis of distributed systems which we consider worth investigating and are part of our agenda. First, the pioneering work of

Masson, Mallela, Dahbura and Yang [Mall78, Dahb84., Yang86....., Yang86] in modeling intermittent faults has opened up a fresh and very practical area for future work. Second, the problem of fault diagnosis in distributed systems with dynamic failure and repair which are not so easily modeled using the PMC method deserves attention. One weakness of the PMC model is the underlying assumption that the test results are available *simultaneously* to a "global observer" who is an unmodeled entity not subject to faults. This runs counter to fundamental principles of distributed computing, where there is no fault-free, omniscient supervisor. Only very recently has research been aimed at producing systems in which diagnosis is performed by the modeled units themselves ([Holt85, Hoss84]) and the outlook is promising.

References

Kime80.

Charles R. Kime, Craig S. Holt, John A. McPherson, and James E. Smith, "Fault diagnosis of distributed systems," *Compsac*, pp. 355-364, 1980.

Prep67.

Franco P. Preparata, Gernot Metze, and Robert T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. Electronic Comput.*, vol. EC-16, pp. 848-854, Dec., 1967.

Bars76.

Ferruccio Barsi, Fabrizio Grandoni, and Piero Maestrini, "A theory of diagnosability of digital systems," *IEEE Trans. Comput.*, vol. C-25, pp. 585-593, June 1976.

Haki74.

S. L. Hakimi and A. T. Amin, "Characterization of connection assignment of diagnosable systems," *IEEE Trans. Comput.*, pp. 86-88, Jan., 1974.

Sull84.Gregory F. Sullivan, "A polynomial time algorithm for fault diagnosability," *Proc. 25th Annual Symp. on Foundations of Computer Science*, pp. 148-156, IEEE Computer Society Publications, Oct., 1984.

Alla75.F. J. Allan, T. Kameda, and S. Toida, "An approach to the diagnosability analysis of a system," *IEEE Trans. Comput.*, pp. 1040-1042, Oct., 1975.

Prep68.

Franco P. Preparata, "Some results on sequentially diagnosable systems," *Proc. of Hawaii Int. Conf. Syst.*, pp. 623-626, 1968.

Karu79.

S. Karunanithi and Arthur D. Friedman, "Analysis of digital systems using a new measure of system diagnosis," *IEEE Trans. Comput.*, vol. C-28, pp. 121-133, Feb., 1979.

Chwa81.

Kyung-yong Chwa and S. Louis Hakimi, "On fault identification in diagnosable systems," *IEEE Trans. Comput.*, vol. C-30, pp. 414-422, June 1981.

Gare79.

M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, San Francisco, CA, 1979.

Mahe76.

Shachindra N. Maheshwari and S. Louis Hakimi, "On models for diagnosable systems and probabilistic fault diagnosis," *IEEE Trans. Comput.*, vol. C-25, pp. 228-236, March 1976.

Fuji78.Hideo Fujiwara and Kozo Kinoshita, "On the computational complexity of system diagnosis," *IEEE Trans. Comput.*, vol. C-27, pp. 881-885, Oct., 1978.

Soma86.

A. Somani, D. Avis, and V. Agarwal, *On complexity of diagnosability and diagnosis problems in system-level diagnosis*, 1986.

Kame75.

T. Kameda, S. Toida, and F. J. Allan, "A diagnosing algorithm for networks," *Information and control*, vol. 29, pp. 141-148, 1975.

Corl76.

A. M. Corluhan and S. L. Hakimi, "On an algorithm for identifying faults in a t-diagnosable system," *Proc. 1976 Conf. Inform. Sci. Syst.*, pp. 370-375, Dep. Elec. Eng., John Hopkins Univ., April 1976.

Madd77.

R. F. Madden, "On fault-set identification in some system-level diagnostic models," *Proc. 1977 Symp. on Fault Tolerant Comp.*, p. 204, June 1977.

Dahb84.

Anton T. Dahbura and Gerald M. Masson, "An $O(n^{2.5})$ fault identification algorithm for diagnosable systems," *IEEE Trans. Comput.*, vol. C-33, pp. 486-492, June 1984.

Yang86.

Che-liang Yang, Gerald M. Masson, and Richard A. Leonetti, "On fault isolation and identification in t_i/t_i- diagnosable systems," *IEEE Trans. Comput.*, vol. C-35, pp. 639-643, July 1986.

Ragh86.

Vijay Raghavan, Saikeung Dong, and Anand Tripathi, "A Characterization of Sequential and t/s-diagnosable systems," *University of Minnesota Tech. Report TR 86-49*, October 1986.

Mall78.

Sivanarayana Mallela and Gerald M. Masson, "Diagnosable systems for intermittent faults," *IEEE Trans. Comput.*, vol. C-27, pp. 560-566, June 1978.

Dahb84.

Anton T. Dahbura, "Fault diagnosis in multiprocessor systems," *Ph.D. Thesis*, Johns Hopkins University, Baltimore, Maryland, 1984.

Yang86.

Che-Liang Yang, "Fault identification and isolation in multiprocessor systems," *Ph.D. Thesis*, Johns Hopkins University, Baltimore, Maryland, 1986.

Yang86.

Che-liang Yang and Gerald M. Masson, "A fault identification algorithm for t_i- diagnosable systems," *IEEE Trans. Comput.*, vol. C-35, pp. 503-509, June 1986.

Holt85.

Craig S. Holt and James E. Smith, "Self-diagnosis in distributed systems," *IEEE Trans. Comput.*, vol. 34, pp. 19-32, Jan., 1985.

Hoss84.

S. H. Hosseini, Jon G. Kuhl, and Sudhakar M. Reddy, "A diagnosis algorithm for distributed computing systems with dynamic failure and repair," *IEEE Trans. Comput.*, vol. 33, pp. 223-233, March 1984.

Towards an improved diagnosability algorithm

Vijay Raghavan
Anand Tripathi

Computer Science Department
136 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

1. Introduction

1.1. The PMC model

In the two decades since the introduction of the so-called PMC model [Prep67], significant progress has been made in the development of the theory associated with the model. The reasons for this are not difficult to find: first, the simplicity and elegance of this model have made it appealing to both theoretical scientists and system designers alike. Second, the universality of the model makes it suitable for capturing the essence of diagnosis in a variety of distributed systems as well as VLSI systems. Third, the model permits a level of abstraction where the behavior of processors/software modules is limited to just two states: faulty or non-faulty. While this may not be the best picture of "real-world" processors and modules, which are inclined to a more fuzzy behavior between the two extremes of being faulty and non-faulty, this simplification has the beneficial effect of enabling diagnostic algorithms developed for the PMC model to be applicable in all layers of the system architecture. Indeed, one may drop the distinctions between hardware processors and software modules and speak only of *units* when modeling the system for fault diagnosis.

A distributed or a multiprocessor system is one in which computational tasks are performed by multiple units. In the PMC model, two principal assumptions are made about the system: first, the variables that characterize the fault behavior of units – Mean Time Between Failure, Mean Time To Repair *inter alia*– are assumed to be *independent* random variables; second, the units are assumed to have the capability to administer tests among themselves for diagnosis. We concern ourselves neither with the precise nature of these tests nor with how and when they are administered, beyond insisting that they be *complete*, i.e., a fault-free unit should always correctly identify the units it tests as being faulty or fault-free.

The system that is to be diagnosed is partitioned into logical units. These units need not be similar in their functionality within the distributed system, except in being able to test singly or in combination, another unit. The outcome of the tests may be classified simply as "pass" or "fail", indicating that the testing unit evaluates the tested unit as being fault-free or faulty, respectively. We assume that the evaluation is significant only if the testing unit itself is fault-free, otherwise the outcome is unreliable. This assumption is known as the *symmetric invalidation* assumption and forms the basis of many offshoots of the PMC model. An alternative assumption, which is also frequently used (and considered by some to represent real systems more closely) is the *asymmetric invalidation* assumption, wherein a "pass" outcome necessarily implies that the tested unit is fault free, but a "fail" outcome only implies that either the tested unit or the testing unit or both are faulty. The rationale behind this assumption [Bars76, Kime80] is that a complete test in systems composed of complex units entails the checking of many responses from the tested system. Therefore, it is extremely unlikely that the faults in the units performing the test would completely cancel the faults in the unit under test, causing a test to pass when it should have failed.

The test system is modeled as a directed graph $G(V,E)$, with the units represented by vertices in the graph and the tests by directed edges. Thus if (i,j) is a directed edge from unit u_i to unit u_j, the unit u_i tests the unit u_j. This directed graph is called the *connection assignment* of the system. A more or less natural measure of the "diagnosability" of a connection assignment is the following:

A system of n units is *one-step t-fault diagnosable* (or simply *t-diagnosable*) if all faulty units within the system can be identified without replacement provided the number of faulty units present does not exceed t.

The diagnosability problem, then, is to identify the *largest t* for which a given connection assignment remains t-diagnosable. Sullivan[Sull84], in a remarkable *tour de force*, presented the first polynomial time algorithm for the diagnosability problem. In what follows, we we develop some concepts which, though not completely used in our present version of our algorithm, are likely to improve the complexity even further. In a forthcoming report, we use some of the concepts presented here to get an $O(|E|^{\frac{5}{2}}|V|^{\frac{-3}{2}})$ algorithm for t-diagnosability.

1.2. Preliminaries: Definitions and notation

For a given connection assignment, represented by a digraph $G(V,E)$, the *diagnosability number* τ_G is the largest non-negative integer t for which G is t-diagnosable. Where the digraph is clear from the context, we may drop the subscript in τ_G. A total function $\sigma:E \rightarrow \{0,1\}$ is said to be a *syndrome* for G. The intuitive idea behind this definition is that since every directed edge represents a "test" in the system being modeled, the test results (or the "syndrome") must simply be a collection of "passes" and "fails", each edge having exactly one of the 2 possibilities. In our notation, a 0 signifies a pass and a 1 a fail.

$F \subseteq V$ is a *consistent fault set* for the syndrome σ if neither

(a) $\sigma(u,v) = 0$ where $u \in V-F$ & $v \in F$ nor

(b) $\sigma(u,v) = 1$ where $u,v \in V-F$

holds.

$F_\sigma = \{F : F$ is a consistent fault set for $\sigma\}$.

$F_{\sigma,t} = \{F : F \in F_\sigma \,\&\, |F| \leq t\}$.

The definition of $F_{\sigma,t}$ allows the following observations: first, a syndrome σ occurs in a t-fault situation if and only if $F_{\sigma,t} \neq \emptyset$. Second, the earlier definition of the diagnosability measure may be restated as follows:

$G(V,E)$ is t-diagnosable if and only if for every syndrome σ for G in a t-fault situation, $|F_{\sigma,t}| = 1$.

Let $v \in V$ be any vertex of G. $d_{in}(v)$ and $d_{out}(v)$ denote, respectively, the in and out degree of the vertex v. The set of vertices of G from which there are directed edges to v is denoted by $\Gamma^{-1}v$, i.e.,

$$\Gamma^{-1}v = \{u : u \in V \;\&\; \langle u,v \rangle \in E\}$$

Also,

$$\Gamma v = \{u : u \in V \;\&\; \langle v,u \rangle \in E\}$$

Let $X \subseteq V$ be some set of vertices, then

$$\Gamma^{-1}X = \bigcup_{v \in X} \Gamma^{-1}v - X$$

$$\Gamma X = \bigcup_{v \in X} \Gamma v - X$$

When we are dealing with more than one digraph, we sometimes use the notations Γ_G^{-1} and Γ_G to avoid ambiguity. The operators Γ^{-1} and Γ take precedence over union and intersection. Thus $\Gamma^{-1}X \cup Y = (\Gamma^{-1}X) \cup Y$.

A *tournament* T_n on n vertices, is a digraph in which every pair of vertices u, v contributes exactly one of (u,v) and (v,u) to the set of edges, i.e., there are no directed cycles of length 2 and the total number of edges is $\frac{n(n-1)}{2}$. A *weighted* tournament is a tournament in which every vertex is assigned a positive weight.

2. Properties of minimal bottleneck sets

The following theorem provides a completely graph-theoretical characterization of t-diagnosability. For a proof, see the referenced papers.

Theorem 1 [Sull84, Alla75]

$G(V,E)$ is t-diagnosable if and only if $\forall Z \subseteq V\ [(Z \neq \emptyset) \Rightarrow \frac{|Z|}{2} + |\Gamma^{-1}Z| > t]$. □

Let τ be the diagnosability number of G. Theorem 1 implies that there exists a non-empty set $Z \subseteq V$ such that $\left\lceil \frac{|Z|}{2} \right\rceil + |\Gamma^{-1}Z| = \tau+1$. This motivates the following definition.

Definition A *bottleneck set* of $G(V,E)$ is a set $Z \subseteq V$ which satisfies $\left\lceil \frac{|Z|}{2} \right\rceil + |\Gamma^{-1}Z| = \tau+1$, where τ is the diagnosability number of G. A *minimal* bottleneck set is a bottleneck set, no smaller subset of which is also a bottleneck set. The *bottleneck function*, Φ_G, of a set $X \subseteq V$ is defined by $\Phi_G(X) = \left\lceil \frac{|X|}{2} \right\rceil + |\Gamma^{-1}X|$. When the context permits no ambiguity, we will drop the subscript in Φ_G.

Theorem 2 The cardinality of a minimal bottleneck set is either 1 or even.

Proof:- Suppose not. Let $Z \subseteq V$, for some digraph $G(V,E)$, be a minimal bottleneck of odd (≥3) cardinality. Let $v \in Z$ be any vertex. Then $Z' = Z - \{v\}$ satisfies $\Phi(Z') = \left\lceil \frac{|Z'|}{2} \right\rceil + |\Gamma^{-1}Z'| \leq \left\lceil \frac{|Z|}{2} \right\rceil + |\Gamma^{-1}Z| = \Phi(Z)$, which contradicts the minimality of Z. □

Theorem 3 Let Z be a minimal bottleneck set of $G(V,E)$. Then the subgraph induced by Z is strongly connected.

Proof:- Suppose not. Then there is a strongly connected component Z' in the subgraph induced by Z, which is a proper subset of Z and has no arcs entering it from within Z. Therefore, $\Phi(Z') = \left\lceil \frac{|Z'|}{2} \right\rceil + |\Gamma^{-1}Z'| \leq \left\lceil \frac{|Z'|}{2} \right\rceil + |\Gamma^{-1}Z| \leq \left\lceil \frac{|Z|}{2} \right\rceil + |\Gamma^{-1}Z| = \Phi(Z)$, which is a contradiction. □

In what follows, we show that a minimal bottleneck set satisfies a property which is stronger than strong connectivity. We call this property "collapsibility".

Definition $Z \subseteq V$ is *collapsible* iff

(a) $|Z| = 1$, or

(b) Z is the union of 2 collapsible sets X and Y such that

$X \cap \Gamma^{-1}Y \neq \emptyset$ and

$Y \cap \Gamma^{-1}X \neq \emptyset$.

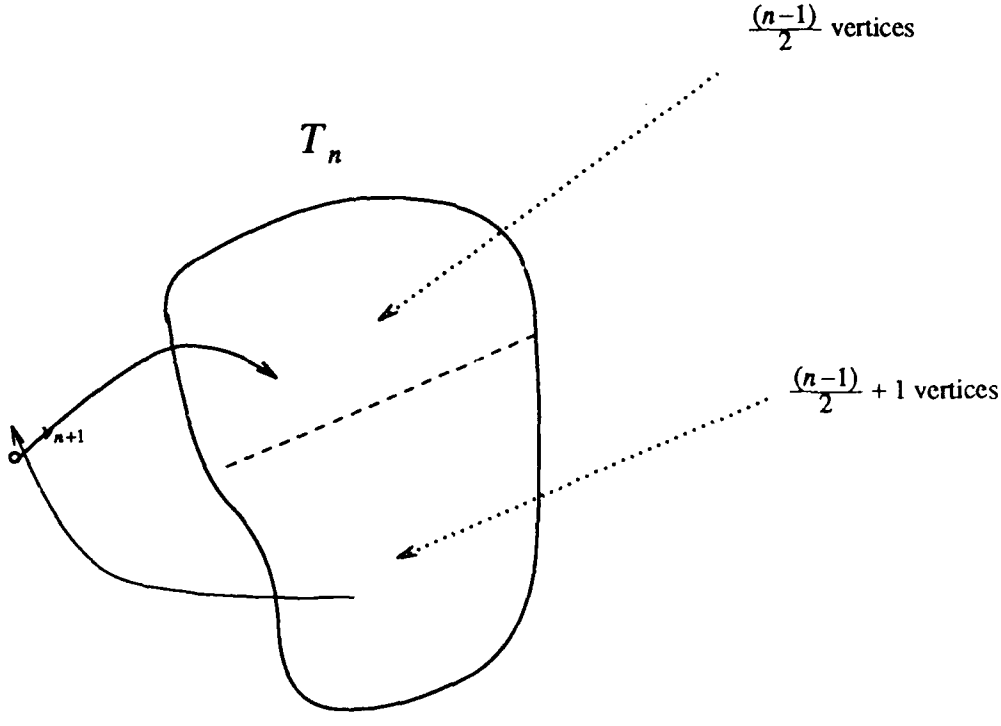
We need a few lemmas to prove that a minimal bottleneck set is collapsible.

Lemma 1:- $\forall n, n \geq 1$, there exists a tournament T_n on n vertices which satisfies:

(a) If n is odd, $\forall v \in V(T_n), d_{in}(v) = d_{out}(v) = \frac{n-1}{2}$.

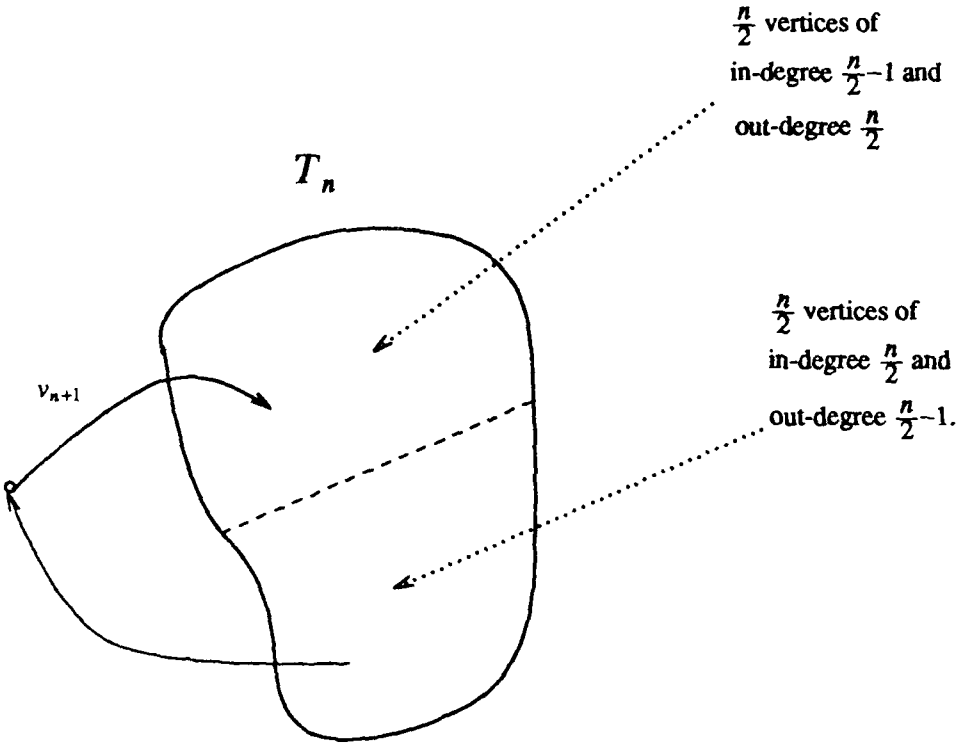
(b) If n is even, there exist $\frac{n}{2}$ vertices v such that $d_{in}(v) + 1 = d_{out}(v) = \frac{n}{2}$ and $\frac{n}{2}$ vertices v such that $d_{in}(v) = d_{out}(v) + 1 = \frac{n}{2}$.

Proof:- By induction. As basis, observe that (a) holds trivially for $n = 1$. Assume that (a) and (b) hold for all $j \leq n$.



Case 1 $n+1$ is even. Then n is odd. Build T_{n+1} as follows. By the induction hypothesis, there exists a tournament T_n on n vertices which satisfies (a). Add a vertex v_{n+1} to this tournament. Connect v_{n+1} to $\frac{n-1}{2}$ vertices of T_n and from the remaining $\frac{n-1}{2}+1$. Then the resulting graph has $\frac{n+1}{2}$ vertices of in-degree $\frac{n+1}{2}$ and out-degree $\frac{n-1}{2}$, and $\frac{n+1}{2}$ vertices with in-degree $\frac{n-1}{2}$ and out-degree

$\frac{n+1}{2}$ which satisfies (b).



Case 2 $n+1$ is odd. By the induction hypothesis, we have a tournament T_n which satisfies (b). Build T_{n+1} by connecting an extra vertex v_{n+1} to all the vertices of T_n which have in-degree of $\frac{n}{2}-1$ and from all the vertices which have out-degree of $\frac{n}{2}-1$. The resulting tournament on $n+1$ vertices satisfies (a), since all the vertices now have in- and out-degree of $\frac{n}{2}$. □

Definition A *balanced* tournament is a tournament which satisfies (a) and (b) of Lemma 1.

Lemma 2:- Let T_n be a tournament on n vertices. Then,

(a) If n is odd $\exists v \in V(T_n)\ [\frac{1}{2} + |\Gamma^{-1}v| \leq \frac{n}{2}]$

(b) If n is even, $\exists v \in V(T_n)\ [\frac{1}{2} + |\Gamma^{-1}v| \leq \frac{n-1}{2}]$

Proof:-

$$\sum_{v \in V(T_n)} d_{in}(v) = \sum_{v \in V(T_n)} |\Gamma^{-1}v| = \frac{n(n-1)}{2}$$

$$\Rightarrow \exists v \in V(T_n) \mid \Gamma^{-1}v \mid \leq \frac{(n-1)}{2}$$

If n is even, the inequality above is strict and the lemma follows. □

Lemma 3:- let T_n be a weighted tournament on n vertices with weight function $\omega: V \rightarrow Z^+$. Then, $\exists v \in V(T_n)$ $[\frac{\omega(v)}{2} + \omega(\Gamma^{-1}v) \leq \frac{\omega(V)}{2}]$.

Proof:- By contradiction. Let T_n be a tournament with weight function ω such that $\forall v \in V(T_n)$ $[\frac{\omega(v)}{2} + \omega(\Gamma^{-1}v) > \frac{\omega(V)}{2}]$. Let $M = \omega(V)$. Number the vertices $v_1, v_2, v_3, \cdots, v_n$ and let their weights be, respectively, $a_1, a_2, a_3, \cdots, a_n$. Then we have $\sum_{i=1}^{n} a_i = M$ and

$$\forall i \quad 1 \leq i \leq n \quad \frac{a_i}{2} + \omega(\Gamma^{-1}v) > \frac{M}{2} \tag{A}$$

Build a tournament T_{2M} on $2M$ vertices

$v_{11}, v_{12}, v_{13}, \cdots, v_{1,a_1}, v_{1,a_1+1}, \cdots v_{1,2a_1}$

$v_{21}, v_{22}, v_{23}, \cdots, v_{2,a_2}, v_{2,a_2+1}, \cdots v_{2,2a_2}$

$\cdots$

$\cdots$

$\cdots$

$v_{n1}, v_{n2}, v_{n3}, \cdots, v_{n,a_n}, v_{n,a_n+1}, \cdots v_{n,2a_n}$

with edges defined by the following two rules :-

(a) If $i \neq j$, then $\forall k, l \quad 1 \leq k \leq 2a_i, \ 1 \leq l \leq 2a_j \quad (v_{ik}, v_{jl}) \in E(T_{2M})$ iff $(v_i, v_j) \in E(T_n)$.

(b) $\forall i \quad 1 \leq i \leq n$, add edges to $E(T_{2M})$ so that the subgraph induced by the row $v_{i1}, v_{i2}, \cdots, v_{i,2a_i}$ is a balanced tournament on $2a_i$ vertices.

By lemma 2(b), $\exists v \in V(T_{2M})$ $[\frac{1}{2} + \mid \Gamma^{-1}_{T_{2M}} v \mid \leq M - \frac{1}{2}]$. Without loss of generality, let v_{11} be a vertex which satisfies this inequality.

Then,

$$\mid \Gamma^{-1}_{T_{2M}} v_{11} \mid \geq 2\omega(\Gamma^{-1}_{T_n} v_1) + a_1 - 1$$

$$\Rightarrow 2\omega(\Gamma^{-1}_{T_n} v_1) + a_1 - 1 \leq M - 1$$

$$\Rightarrow \omega(\Gamma^{-1}_{T_n} v_1) + \frac{a_1}{2} \leq \frac{M}{2}$$

which contradicts (A). □

Theorem 4:- Every minimal bottleneck of G is collapsible.

Proof:- Suppose not. Let $X \subseteq V(G)$ be a minimal bottleneck which is not collapsible. Then X can be expressed as the union of $m \geq 2$ disjoint collapsible sets $X_1, X_2, \cdots, X_m$ such that

$$\forall i,j \quad i \neq j \Rightarrow \Gamma^{-1}X_i \cap X_j = \emptyset \text{ or } \Gamma^{-1}X_j \cap X_i = \emptyset$$

Build a weighted tournament T_m on m vertices $v_1, v_2, \cdots v_m$ with weight function ω as follows:-

$$\omega(v_i) = |X_i| \qquad 1 \leq i \leq m$$

The edges of this tournament, $E(T_m)$ are built as follows:-

For every pair of vertices, v_i and v_j, exactly one of (v_i, v_j) or (v_j, v_i) must belong to $E(T_m)$. If $\Gamma^{-1}X_i \cap X_j \neq \emptyset$, then let (v_j, v_i) belong to $E(T_m)$, else if both $\Gamma^{-1}X_i \cap X_j$ and $\Gamma^{-1}X_i \cap X_i$ are empty, let (v_i, v_j) belong to $E(T_m)$ if and only if $i < j$. Clearly, the edges as defined above will produce a tournament on m vertices.

By lemma 3,

$$\exists v \in V(T_m) \; [\frac{\omega(v)}{2} + \omega(\Gamma_{T_m}^{-1}v) \leq \frac{\omega(V(T_m))}{2}]$$

$$\Rightarrow \exists i \;\; 1 \leq i \leq m \; [\frac{|X_i|}{2} + |\Gamma^{-1}X_i| \leq \frac{|X|}{2} + |\Gamma^{-1}X|]$$

$$\Rightarrow \exists i \;\; 1 \leq i \leq m \; [\Phi(X_i) \leq \Phi(X)]$$

which contradicts the minimality of X. □

Corollary [Haki74] If $G(V,E)$ is a digraph with no directed cycles of length 2, then the diagnosability number of G is the minimum in-degree of any vertex in G.

Proof:- If there are no directed cycles of length 2, then the largest collapsible set has cardinality 1. Therefore, every minimal bottleneck set consists of a single vertex and the bottleneck function for minimal bottlenecks is minimized when a vertex of minimum in-degree is chosen. □

Note 1:- In the proof of the above theorem, we are generous in adding edges to get a tournament on m vertices even when the original graph may not have had such edges. This raises the question of whether the theorem can be improved to produce the following claim:

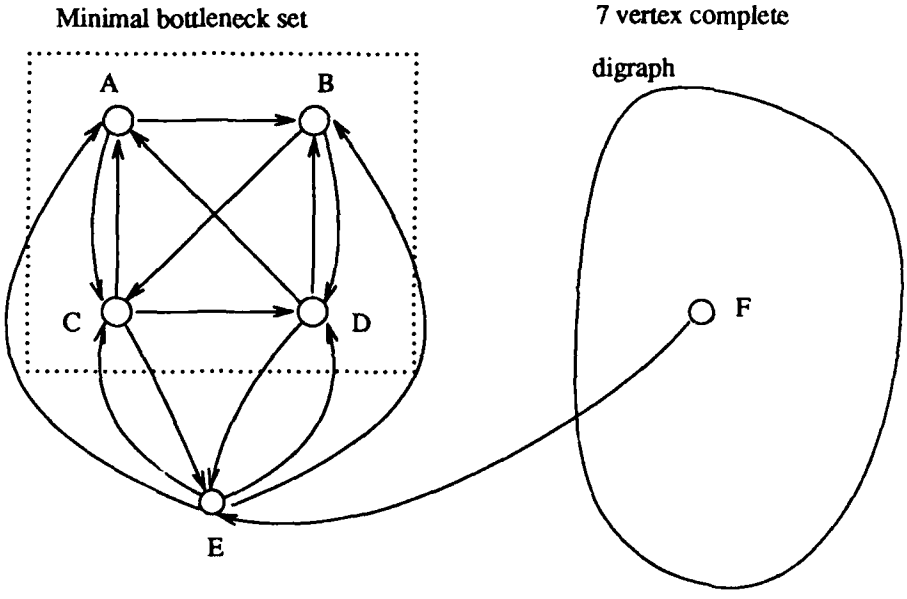
A minimal bottleneck is strongly collapsible, where strong collapsibility is defined by:

Definition $X \subseteq V(G)$ is *strongly collapsible* iff

(a) $|X| = 1$, or

(b) X can be partitioned into 2 strongly collapsible sets Y and Z such that $\exists v \in Y, \exists z \in Z\ [(v,z),(z,v) \in E(G)]$.

That a minimal bottleneck set need not be strongly collapsible is seen by the following example:



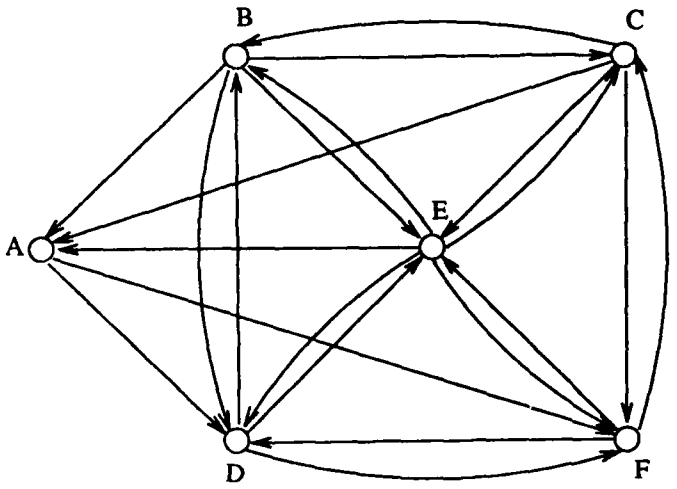
Here, the unique minimal bottleneck set $\{A, B, C, D\}$ is not strongly collapsible since the only partition into two strongly collapsible sets does not satisfy (b) of the definition. □

Note 2:- Can Theorem 4, however, be strengthened as follows:

Let Z be a minimal bottleneck set and let $v \in Z$ be some vertex in Z. Then there is a collapsing sequence starting with v which collapses Z, where we define a collapsing sequence as follows:

Definition A *collapsing sequence* is a sequence of m vertices $v_1, v_2, v_3, \cdots, v_m$ such that $\forall i\ 2 \le i \le m \quad v_i \in \Gamma^{-1}\{v_1, v_2, \cdots, v_{i-1}\}$ and $\Gamma^{-1}v_i \cap \{v_1, v_2, \cdots, v_{i-1}\} \neq \emptyset$. Here m is the *length* of the collapsing sequence. We say that a vertex $v \in Z$ *collapses* Z if there is a collapsing sequence starting with v of length $|Z|$ which consists only of vertices in Z.

It should be clear that not all collapsible sets need have the property that they are collapsible from every vertex contained in them. In fact, one can quite easily build collapsible sets which do not contain even a single vertex from which the entire set may be collapsed. Unfortunately, it turns out that minimal bottleneck sets may have vertices from which they cannot be collapsed, as the following example illustrates:



In the above digraph, the minimal bottleneck set is again unique, and consists, in fact, of the entire set of vertices $\{A, B, C, D, E, F\}$. However, there is no collapsing sequence starting with A which collapses the set. □

We do succeed in proving (Theorem 6) a weaker form of the claim envisaged in the above note, *viz.*, that a minimal bottleneck set consists of at least one vertex from which it may be collapsed. Theorem 5 below is a refinement of Theorem 3. We show that the distance between any 2 vertices in a minimal bottleneck set must be quite small.

Definition The *distance* from vertex u to vertex v, denoted by $dist(u,v)$ is the length of the shortest directed path from u to v. If no path exists, then $dist(u,v) = \infty$.

Theorem 5:-

Let $u, v \in Z$, where $Z \subseteq V(G)$ is a minimal bottleneck set. Then $dist(u,v) \leq 1 + \log_2(2\tau + 1)$, where τ is the diagnosability number of G.

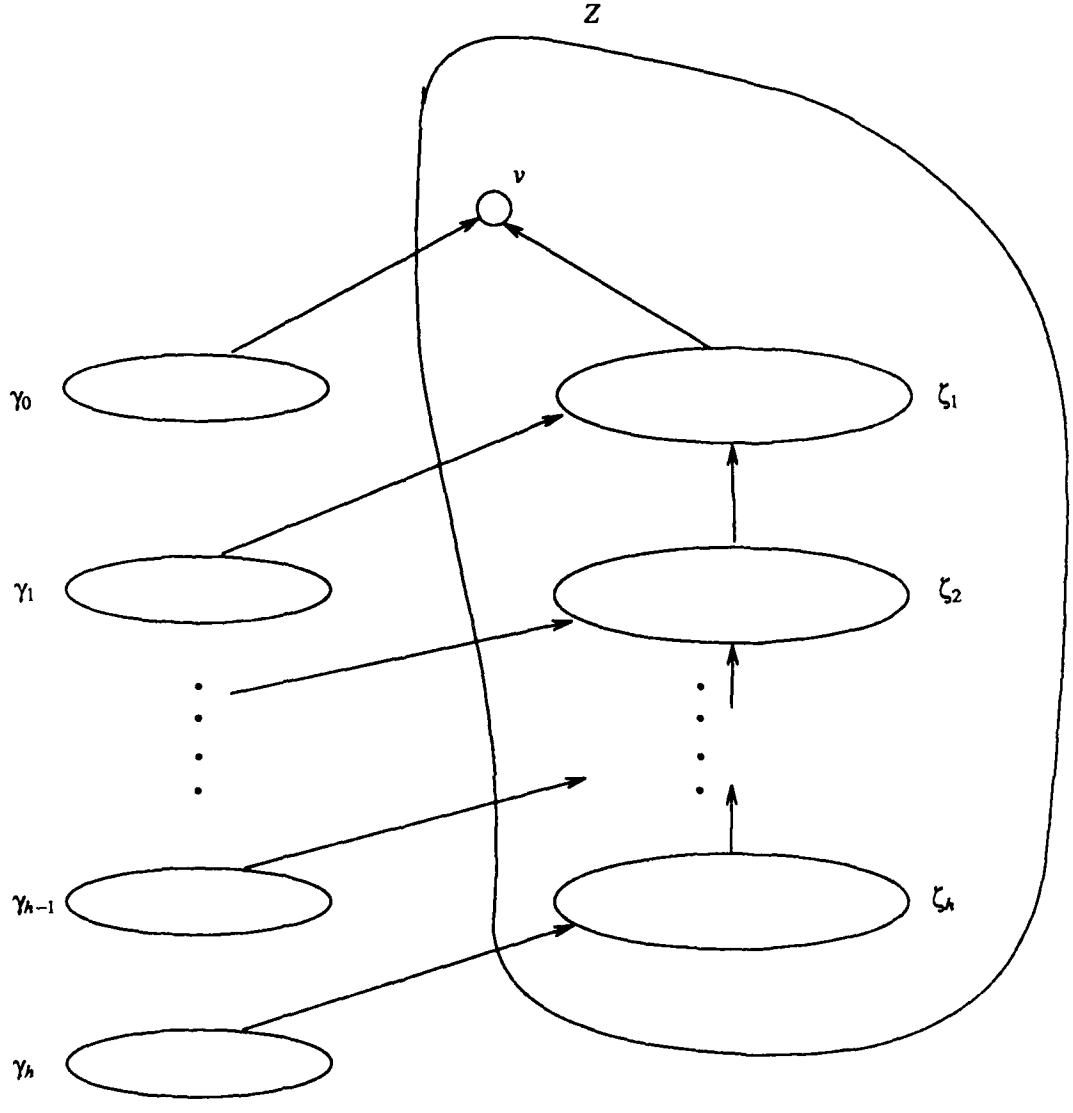
Proof:- Let $\zeta_i = \{x : x \in Z \ \& \ dist(x,v) = i\}$. Thus $\zeta_0 = \{v\}$. Let γ_i be defined by

$$\gamma_0 = \Gamma^{-1}v - Z$$

and for $i > 0$,

$$\gamma_i = \Gamma^{-1}\bigcup_{j=0}^{i}\zeta_j - \Gamma^{-1}\bigcup_{j=0}^{i-1}\zeta_j - Z$$

(Look at picture below.)



Let h be the least number such that $\zeta_h \neq \emptyset$. Then $Z = \bigcup_{i=0}^{h} \zeta_i$ and $\Gamma^{-1}Z = \bigcup_{i=0}^{h} \gamma_i$.

By the minimality of Z, we get the following h inequalities:

$$\left\lceil \frac{|\zeta_0|}{2} \right\rceil + |\gamma_0| + |\zeta_1| > \tau + 1$$

$$\left\lceil \frac{|\zeta_0 \cup \zeta_1|}{2} \right\rceil + |\gamma_0| + |\gamma_1| + |\zeta_2| > \tau + 1$$

...

...

$$\left\lceil \frac{|\bigcup_{i=0}^{h-1} \zeta_i|}{2} \right\rceil + |\bigcup_{i=0}^{h-1} \gamma_i| + |\zeta_h| > \tau + 1$$

Substituting $\tau + 1 = \left\lceil \frac{|Z|}{2} \right\rceil + |\Gamma^{-1}Z| = |\bigcup_{i=0}^{h} \zeta_i| + |\bigcup_{i=0}^{h} \gamma_i|$ in the above inequalities and rearranging terms, we get

$$|\zeta_1| > \left\lfloor \frac{|\bigcup_{i=1}^{h} \zeta_i|}{2} \right\rfloor + |\bigcup_{i=1}^{h} \gamma_i|$$

$$|\zeta_2| > \left\lfloor \frac{|\bigcup_{i=2}^{h} \zeta_i|}{2} \right\rfloor + |\bigcup_{i=2}^{h} \gamma_i|$$

...

...

$$|\zeta_h| > \left\lfloor \frac{|\zeta_h|}{2} \right\rfloor + |\gamma_h|$$

Since $a > \left\lfloor \frac{b}{2} \right\rfloor + c \Rightarrow 2a > b + 2c$, we have,

$$|\zeta_1| > |\bigcup_{i=2}^{h} \zeta_i| + 2|\bigcup_{i=1}^{h} \gamma_i|$$

$$|\zeta_2| > |\bigcup_{i=3}^{h} \zeta_i| + 2|\bigcup_{i=2}^{h} \gamma_i|$$

...

...

$$|\zeta_h| > 2|\gamma_h|$$

Unfolding the recurrence inequalities backwards and allowing for $|\gamma_i| = 0, \quad 1 \le i \le h$, we have,

$$|\zeta_h| = |\zeta_h|$$

$$|\zeta_{h-1}| \ge |\zeta_h| + 1$$

$$|\zeta_{h-2}| \ge 2|\zeta_h| + 2$$

$$|\zeta_{h-3}| \ge 2^2|\zeta_h| + 3$$

$$\ldots$$

$$\ldots$$

$$|\zeta_1| \ge 2^{h-2}|\zeta_h| + (h-1)$$

$$|\zeta_0| = |\{v\}| = 1$$

Since

$$2\tau = 2\left(\left\lceil \frac{|Z|}{2} \right\rceil + |\Gamma^{-1}Z| - 1\right)$$

$$\ge |Z| + 2|\Gamma^{-1}Z| - 2$$

$$\ge |Z| - 2$$

$$\ge 2^{h-1}|\zeta_h| + 1 + \frac{(h-1)h}{2} - 2$$

we get

$$\frac{2\tau + 1}{|\zeta_h|} \ge 2^{h-1} + \frac{h(h-1)}{2|\zeta_h|}$$

$$h \le \log_2\left(\frac{2\tau + 1}{|\zeta_h|}\right) + 1$$

Since $|\zeta_h| \ge 1$, we have the desired inequality. □

Definition $X \subseteq V(G)$ is *sequentially* collapsible if there exists a vertex $v \in X$ which collapses X.

We need the following lemma to prove that a minimal bottleneck set is sequentially collapsible.

Lemma 4 Let $G(V,E)$ be a digraph such that $V = \bigcup_{i=1}^{m} X_i$, where each of the X_is is neither empty nor equal to V and

$$\forall i \quad 1 \le i \le m \quad \Gamma^{-1}X_i \cap \Gamma X_i = \emptyset \tag{A}$$

holds.

Then for some X_i, $1 \le i \le m$,

$$\exists Y \subseteq X_i \quad [Y \neq \emptyset \ \& \ \Phi(Y) \le \left\lceil \frac{|V|}{2} \right\rceil]$$

Proof The proof is by induction on the cardinality of V. As basis, observe that if $|V| = 1$, the antecedent of the lemma can never be satisfied since it is impossible to find a non-empty subset of a 1-element set which is not equal to the set. Therefore, the lemma is vacuously true in this case. Assume that it is true whenever $|V| \le n$. We must show that it holds for $|V| = n+1$.

Case 1 $n+1$ is odd.

Consider the subgraph $G_1(V_1,E_1)$ induced by deleting some vertex $v \in V$. Now $|V_1| = |V| - 1 = n$ is even. If there exists some X_i $1 \le i \le m$ such that $X_i = V_1$, then certainly $\Phi_G(X_i) = \frac{n}{2} + |\Gamma^{-1}X_i| \le \frac{n}{2} + 1 = \left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{|V|}{2} \right\rceil$, and the lemma is proved by letting $Y = X_i$.

If there is no such X_i, then since $V_1 = \bigcup_{i=1}^{m} X_i'$, where $X_i' = X_i - \{v\}$, and since (A) holds in the induced subgraph as well, by the induction hypothesis there exists a non-empty subset Y of some X_i' such that

$$\Phi_{G_1}(Y) = \left\lceil \frac{|Y|}{2} \right\rceil + |\Gamma_{G_1}^{-1}Y| \le \left\lceil \frac{|V_1|}{2} \right\rceil = \frac{n}{2}$$

But then,

$$\Phi_G(Y) \le \Phi_{G_1}(Y) + 1 \le \frac{n}{2} + 1 = \left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{|V|}{2} \right\rceil$$

and the lemma holds.

Case 2 $n+1$ is even.

(This case is much more difficult than the previous case and the proof is in 4 parts. In the first part, we show that any given digraph which satisfies the antecedent of the lemma can be converted to an equivalent digraph in which an additional key property is satisfied. This property provides the inequality necessary to identify at least one subset Y which will prove the lemma. In part 2, we define an equivalence relation which partitions the given digraph into disjoint equivalence classes of vertices; the rest of the proof is directed toward demonstrating that one of these equivalence classes is, in fact, the Y needed to complete the proof. With this in mind, we transform the digraph so that every vertex in any equivalence class A_k has exactly the same in- and out-neighbors from other equivalence classes as the other vertices in A_k; the sub-graph induced by the equivalence class itself is made a balanced tournament. This has the effect of setting up a correlation between the in-degree of every vertex in the equivalence class A_k, and $\Phi(A_k)$. In part 3, we prepare the ground for the final pigeonholing argument in part 4, which will demonstrate that there must be some vertex which has a small enough in-degree for the equivalence class to which it belongs to satisfy the requirement of the lemma.))

Part 1

In this part, we show how the given graph G may be modified to obtain a new graph in which some key properties are true. More precisely, we have the following claim:

Claim 1

Given a graph G which satisfies the antecedent of the lemma, there exists a graph $G'(V',E')$ which satisfies:

(a) $V' = V$

(b) $V' = \bigcup_{i=1}^{m'} X'_i$, where each of the X'_is is a non-empty set not equal to V', and $m' \geq m$, and

$\forall i \ 1 \leq i \leq m' \ \exists j \ 1 \leq j \leq m \ [X'_i \subseteq X_j]$.

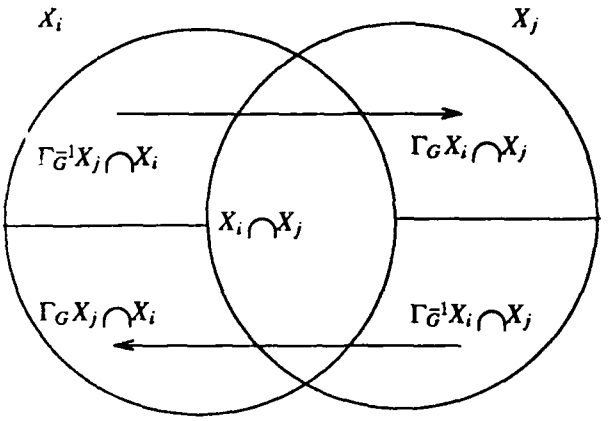
(c) $\forall i \ 1 \leq i \leq m' \ [\Gamma_{\bar{G}} X'_i \cap \Gamma_G X'_i = \emptyset]$.

(d) $\forall i \ 1 \leq i \leq m' \ \forall Y \subseteq X'_i \ [\Phi_{G'}(Y) = \Phi_G(Y)]$.

and

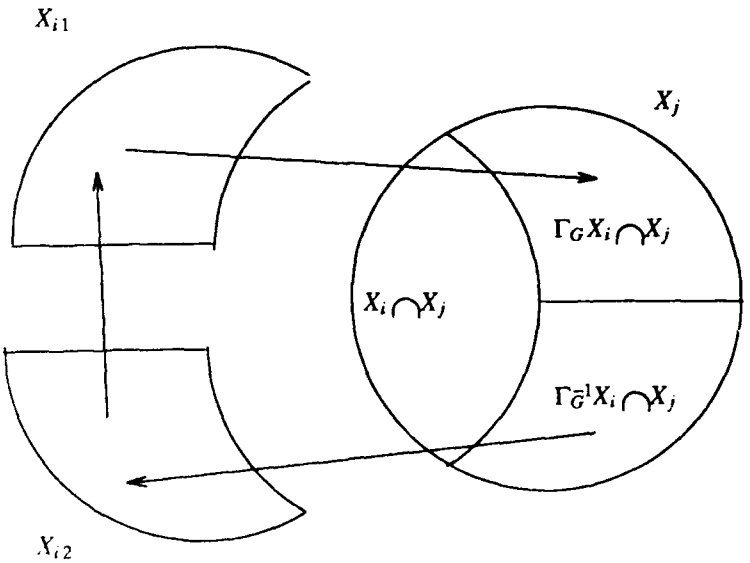
(e) $\forall i, j \ 1 \leq i, j \leq m'$, if $\Gamma_G X'_j \cap X'_i \neq \emptyset$ then $|\Gamma_{\bar{G}} X'_j \cap X'_i| > |\Gamma_G X'_i \cap X'_j|$.

Proof of Claim 1 Note that (a)-(d) are satisfied by setting $X'_i = X_i$ $1 \le i \le m$ and $m' = m$. The only new requirement is (e). If it is already satisified in G, we are done. So suppose that for some pair of sets X_i, X_j $1 \le i, j \le m$, (e) does not hold. Therefore $\Gamma_G X_j \cap X_i \neq \emptyset$ and $|\Gamma_{\bar{G}}^{-1} X_j \cap X_i| \le |\Gamma_G X_i \cap X_j|$. (Look at figure below.)



Clearly, $(\Gamma_{\bar{G}}^{-1} X_j \cap X_i) \cap (\Gamma_G X_j \cap X_i) = \emptyset$ and $(\Gamma_G X_i \cap X_j) \cap (\Gamma_{\bar{G}}^{-1} X_i \cap X_j) = \emptyset$, else (A) of the lemma is not satisfied for G.

Let $X_{i1} = \Gamma_{\bar{G}}^{-1} X_j \cap X_i$ and $X_{i2} = \Gamma_G X_j \cap X_i$. Let $f : X_{i1} \to \Gamma_G X_i \cap X_j$ be any one-one mapping. Such a mapping is possible since $X_{i1} \le |\Gamma_G X_i \cap X_j|$.



Now, for every vertex $v \in X_{i2}$, delete all directed edges from vertices $u \in X_{i1}$ and replace each deleted edge by a new edge from $f(u)$, which is in $\Gamma_G X_i \cap X_j$.

Call the graph thus obtained H_1. Let $m_1 = m + 1$. In H_1, $\Gamma_{H_1} X_{i1} \cap X_{i2} = \emptyset$, so (e) is satisfied between X_{i1} and X_{i2}. Similarly, (e) is satisfied between X_j and X_{i1}, and between X_j and X_{i2}.

It can be verifed that (a)-(d) are still satisfied for all the m_1 sets in H_1.

If X_i, X_j was the only pair of sets which violated (e) in G, then we are done by setting $G' = H_1$ and $m' = m_1$. Else, we repeat the above transformation for some pair of sets in H_1 for which (e) is not satisfied to obtain H_2 and so on. The sequence of graphs $G=H_0, H_1, H_2, \cdots$ obtained by these repeated transformations is finite and the last graph G' in the sequence satisfies all the properties (a)-(e). □

Note that it suffices to prove *case 2* in the transformed graph G' because of properties (b) and (d). For this reason, and in the interests of avoiding avoidable superscripts, we will suppose without loss of generality that the given graph G satisfies properties (a)-(e).

Part 2

Define an equivalence relation $\simeq$ on V as follows:

$u \simeq v$ if and only if for every i $1 \le i \le m$, either both u and v belong to X_i or both u and v do not belong to X_i, and, in addition, if u and v *do* belong to X_i then $\forall j\ 1 \le j \le m\ [(j \ne i) \Rightarrow ((\Gamma_G u \cap X_j \ne \emptyset \Rightarrow \Gamma_{\bar{G}}^{1} v \cap X_j = \emptyset)\ \&\ (\Gamma_G v \cap X_j \ne \emptyset \Rightarrow \Gamma_{\bar{G}}^{1} u \cap X_j = \emptyset))]$.

That $\simeq$ is an equivalence relation may be easily verfied. Being an equivalence relation, $\simeq$ partitions V into a finite number of equivalence classes $A_1, A_2, A_3, \cdots, A_q$ which are pairwise disjoint and do not span set boundaries, i.e, $\forall k\ 1 \le k \le q\ \ \forall i, j\ 1 \le i, j \le m\ \ (i \ne j) \Rightarrow (A_k \cap X_i \ne \emptyset\ \&\ A_k \cap X_j \ne \emptyset\ \Rightarrow\ A_k \subseteq X_i \cap X_j)$.

Build a new graph $G_1(V_1, E_1)$ from G using the following rules:-

(a) $V = V_1$.

(b) The edges in E_1 are built according to one of 2 rules:

(b1) Let A_k, A_l be any 2 distinct equivalence classes induced by $\simeq$ in V. For every pair of vertices u, v such that $u \in A_k$ and $v \in A_l$, let $(u, v) \in E_1$ if and only if $\Gamma_{\bar{G}}^{1} A_l \cap A_k \ne \emptyset$.

(b2) Add enough edges to E_1 so that the subgraph induced by every A_k $1 \le k \le q$ is a balanced tournament.

Note that G_1 as constructed above preserves properties (a), (b), (c), and (e) of claim 1, but (d) may be violated.

Claim 2

Let A_k be any equivalence class induced by $\simeq$ in V and let $v \in A_k$ be some vertex. Let $d_{in}^{G_1}(v)$ denote the in-degree of v in G_1. Then $d_{in}^{G_1}(v) \ge \Phi_G(A_k) - 1$.

Proof of claim 2

$$d_{in}^{G_1}(v) = |\Gamma_{\bar{G}_1}^{1} v|$$

$$= |\Gamma_{\bar{G}_1}^{1} v \cap A_k| + |\Gamma_{\bar{G}_1}^{1} v \cap \bar{A}_k|$$

Now $|\Gamma_{\bar{G}_1}^{1} v \cap A_k| \geq \left\lceil \frac{|A_k|}{2} \right\rceil - 1$ by lemma 1 and $|\Gamma_{\bar{G}_1}^{1} v \cap \bar{A}_k| = \Gamma_{\bar{G}}^{1} A_k$ by the construction (b1) above. Therefore, the claim follows. □

Claim 3

Let $u, v \in V$ be any pair of vertices. Then there exists a 2-cycle between u and v in G_1 only if both the conditions below are satisfied:

(i) For some X_i $1 \leq i < m$, both u and v belong to distinct equivalence classes in X_i. Moreover, then there is a 2-cycle between every pair of vertices x, y where $x \in A_k$ and $y \in A_l$.

(ii) There exists some X_j $1 \leq j \leq m$ such that either $u \in \Gamma_{\bar{G}_1}^{1} X_j$ and $v \in \Gamma_{G_1} X_j$ or $u \in \Gamma_{G_1} X_j$ and $v \in \Gamma_{\bar{G}_1}^{1} X_j$.

Proof of (i): If u and v belong to distinct sets, X_i and X_j, then property (c) of claim 1 is violated for both X_i and X_j. If u and v belong to the same equivalence class A_k, then construction (b2) ensures that there is anly one directed edge between u and v. Therefore u and v must belong to distinct equivalence classes, say A_k and A_l, which are both contained in the same set, say X_i. Moreover, all $x \in A_k$ are equivalent under $\simeq$ to u, and all $y \in A_l$ are equivalent under $\simeq$ to v. By the construction (b1) therefore, there is a 2-cycle between x and y if and only if there is one between u and v.

Proof of (ii): By (i), u, v belong to the same set X_i, but not to the same equivalence class. Therefore, (ii) follows from the definition of the relation $\simeq$. □

Part 3

The aim of this part is to establish that $|E_1| \leq \frac{n(n+1)}{2}$. There seems to be no straightforward pigeonholing argument to prove this and we resort to another construction:

For every pair u, v of vertices in V_1 such that a 2-cycle exists between u and v, color one edge, say (u, v), red and the other edge (v, u) blue. Now repeat the following algorithm for all pair of sets X_i, X_j, $1 \leq i < j \leq m$ in G_1.

Step 1:

Delete all red edges (u, v) such that one of the following is satisfied:

i $u \in \Gamma_{G_1} X_j \cap X_i$ and $v \in \Gamma_{\bar{G}_1^1} X_j \cap X_i$.

ii $u \in \Gamma_{\bar{G}_1^1} X_j \cap X_i$ and $v \in \Gamma_{G_1} X_j \cap X_i$.

iii $u \in \Gamma_{G_1} X_i \cap X_j$ and $v \in \Gamma_{\bar{G}_1^1} X_i \cap X_j$.

iv $u \in \Gamma_{\bar{G}_1^1} X_i \cap X_j$ and $v \in \Gamma_{G_1} X_i \cap X_j$.

Step 2:

Add an edge (u,v) between every pair of vertices (u,v) such that $u \in \Gamma_{\bar{G}_1^1} X_j \cap X_i$ and $v \in \Gamma_{\bar{G}_1^1} X_i \cap X_j$ or $u \in \Gamma_{G_1} X_j \cap X_i$ and $v \in \Gamma_{G_1} X_i \cap X_j$.

Call the graph so obtained $G_2(V_2,E_2)$.

Claim 4 G_2 has no 2-cycles.

Proof of Claim 4 All the red edges must have been deleted in step 1 above because of claim 3(ii). Therefore, all the old 2-cycles in G_1 have been broken. The only new edges added in step 2 are between vertices which in G_1 could not have had any edges between them (else property (c) of claim 1 would have been violated), and even then precisely one edge is added between every such pair. □

Claim 5 $\frac{n(n+1)}{2} \geq |E_2| \geq |E_1|$.

Proof of Claim 5 Let e_{1ij} and e_{2ij} denote, respectively, the number of edges deleted in step 1 and the number of edges added in step 2, for any pair of sets X_i and X_j, $1 \leq i < j \leq m$. Let a, b, c, d, denote, respectively, $|\Gamma_{\bar{G}_1^1} X_j \cap X_i|$, $|\Gamma_{G_1} X_j \cap X_i|$, $|\Gamma_{G_1} X_i \cap X_j|$, $|\Gamma_{\bar{G}_1^1} X_i \cap X_j|$. Then, $e_{1ij} \leq ab + cd$ and $e_{2ij} = ad + bc$.

Now, if b or c is equal to 0, then certainly $e_{1ij} = e_{2ij} = 0$. Else, if both b and c are not equal to 0, then by property (e) of claim 1, which holds for G_1, $a > c$ and $d > b$. Therefore,

$$a(d-b) > c(d-b)$$

or $$ad + bc > cd + ab$$

which implies that $e_{2ij} > e_{1ij}$.

From the above, we can conclude in any case that $e_{2ij} \geq e_{1ij}$ for any pair of sets X_i and X_j.

Therefore,

$$|E_2| = |E_1| + \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} (e_{2ij} - e_{1ij}) \geq |E_1|$$

and since there are no 2-cycles in G_2,

$|E_2| \leq \frac{n(n+1)}{2}$ □

Part 4

Now we finally prove *case 2* of the lemma.

By claim 5, $|E_1| \leq \frac{n(n+1)}{2}$. Also $|E_1| = \sum_{v \in V_1} d_{in}^{G_1}(v)$, so there must exist some vertex $v \in V_1$ such that $d_{in}^{G_1}(v) \leq \frac{n}{2}$. Since $n+1$ is even, and $d_{in}^{G_1}(v)$ must be an integer, we can say that $d_{in}^{G_1}(v) \leq \frac{n-1}{2}$.

Let A_k be the equivalence class to which v belongs. Then, by claim 2, we have $d_{in}^{G_1}(v) \geq \Phi_G(A_k) - 1$. From this and the above, we can conclude that

$$\Phi_G(A_k) - 1 \leq \frac{n-1}{2}$$

$$\text{or, } \Phi_G(A_k) \leq \frac{n+1}{2} = \frac{|V|}{2}$$

The lemma then follows by letting $Y = A_k$. □

Theorem 6 Every minimal bottleneck of any digraph $G(V,E)$ is sequentially collapsible.

Proof Suppose not. Let $Z \subseteq V$ be a minimal bottleneck set which is not sequentially collapsible. For each $v \in Z$, define

$X_v = \{x$: There is a collapsing sequence $v_0, v_1, v_2, \cdots, v_k$, for some $0 \leq k \leq |Z|$, with $v_0 = v$ and $v_k = x$ and $\forall i \ 1 \leq i \leq k \ \ v_i \in Z\}$.

Clearly, $w \in X_v \Rightarrow X_w \subseteq X_v$. Let $X_1, X_2, \cdots, X_m$ be the *maximas* of the partial order defined by $\subseteq$ on the set $\bigcup_{v \in Z} \{X_v\}$, i.e., $\forall i \ 1 \leq i \leq m \ \forall j \ 1 \leq j \leq m \ \ (i \neq j) \Rightarrow (X_i \not\subseteq X_j)$. Then $Z = \bigcup_{i=1}^{m} X_i$ and $\forall i \ 1 \leq i \leq m \ \ \Gamma_G^{-1} X_i \cap \Gamma_G X_i = \emptyset$ (else X_i is not maximal.)

Therefore, the subgraph G_1 induced by Z satisfies the antecedent of lemma 4. Consequently, for some $X_i \ \ 1 \leq i \leq m$, we can find a non-empty subset Y of X_i such that $\Phi_{G_1}(Y) \leq \left\lceil \frac{|Z|}{2} \right\rceil$. But then,

$$\Phi_G(Y) \leq \Phi_{G_1}(Y) + |\Gamma_G^{-1} Z| \leq \left\lceil \frac{|Z|}{2} \right\rceil + |\Gamma_G^{-1} Z|$$

contradicts the minimality of Z. □

References

Prep67.

Franco P. Preparata, Gernot Metze, and Robert T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. Electronic Comput.*, vol. EC-16, pp. 848-854, Dec., 1967.

Bars76.

Ferruccio Barsi, Fabrizio Grandoni, and Piero Maestrini, "A theory of diagnosability of digital systems," *IEEE Trans. Comput.*, vol. C-25, pp. 585-593, June 1976.

Kime80.

Charles R. Kime, Craig S. Holt, John A. McPherson, and James E. Smith, "Fault diagnosis of distributed systems," *Compsac*, pp. 355-364, 1980.

Sull84.Gregory F. Sullivan, "A polynomial time algorithm for fault diagnosability," *Proc. 25th Annual Symp. on Foundations of Computer Science*, pp. 148-156, IEEE Computer Society Publications, Oct., 1984.

Alla75.F. J. Allan, T. Kameda, and S. Toida, "An approach to the diagnosability analysis of a system," *IEEE Trans. Comput.*, pp. 1040-1042, Oct., 1975.

Haki74.

S. L. Hakimi and A. T. Amin, "Characterization of connection assignment of diagnosable systems," *IEEE Trans. Comput.*, pp. 86-88, Jan., 1974.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.